Basic business logic

lesson5

You have made your domain model to run a web application. This application is already a useful one, but still there are a lot of refinements that can be made to it. Let's do what's necessary to convert your application into a better application and, in this way you shall learn some new interesting things about OpenXava.

We'll start adding business logic to your entities in order to convert your application into something more than a simple database manager.

5.1 Calculated properties

Perhaps the most simple business logic you can add to your application is a calculated property. The properties you have used until now are persistent, i.e., each property is stored in a column in a table in the database. A calculated property is a property that does not store its value in the database but it's calculated any time the property is accessed. See the difference between a persistent and a calculated property in listing 5.1.

Listing 5.1 Difference between a persistent property and a calculated property

```
// Persistent property
private int quantity;   // Has a field, so it's persistent
public int getQuantity() {   // A getter to return the field value
    return quantity;
}
public void setQuantity(int quantity) {   // Changes the field value
    this.quantity = quantity;
}

// Calculated property
public int getAmount() {   // It has no field and no setter, only a getter
    return quantity * price;   // with a calculation
}
```

Calculated properties are automatically recognized by OpenXava. You can use them in views, tabular lists or any other part of your code.

We are going to use calculated properties to add the money element to our Invoicing applications. Because, we have details, products, quantities. But what about amounts?

5.1.1 Simple calculated property

The first step will be to add an amount property to the `Detail`. We want the detail amount to be recalculated and shown to the user when the user chooses a product and type in the quantity, just as shown in figure 5.1.

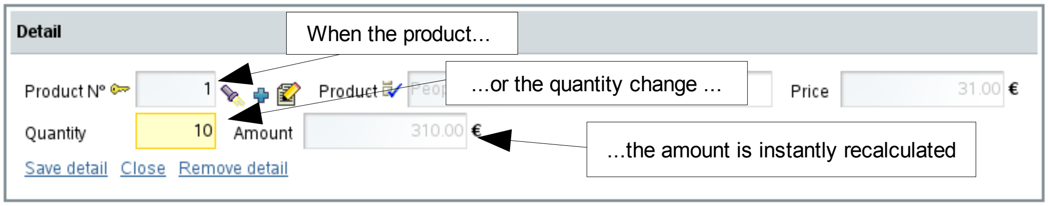


Figure 5.1 Behavior of the calculated property amount in Detail

Adding this feature to your current code is practically adding a calculated property to `Detail`. Just add the code in listing 5.2 to the `Detail` code.

Listing 5.2 Calculated property amount in the Detail class

```
@Stereotype("MONEY")
@Depends("product.number, quantity")      // When  the user changes product or quantity
public BigDecimal getAmount() {           // this property is recalculated and redisplayed
    return new BigDecimal(quantity).multiply(product.getPrice());
}
```

Simply put the calculation in `getAmount()` and use `@Depends` to indicate to OpenXava that the `amount` property depends on `product.number` and `quantity`, thus each time the user changes any of these values the property will be recalculated.

Now you have to add this new property to the view for `Detail` (listing 5.3).

Listing 5.3 Adding the amount property to the Detail view

```
@View(members="product; quantity, amount")  // amount added
public class Detail extends Identifiable {
```

The only remaining detail is to modify the `Simple` view for `Product` in order to show the price. See listing 5.4.

Listing 5.4 Adding price property to the Product view

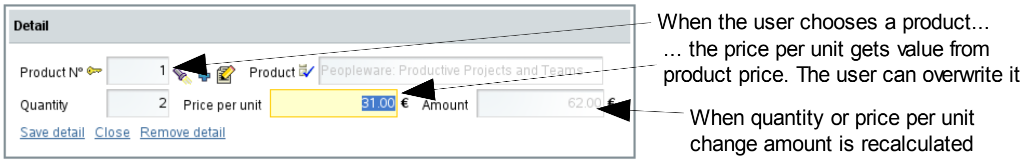
```
@View(name="Simple", members="number, description, price")  // price added
public class Product {
```

Nothing else is required. The mere addition of the getter and modifying the view is enough. Try the `Invoice` and `Order` modules to see the `amount` property in action.

5.1.2 Using @DefaultValueCalculator

The way we calculated the amount for the detail line is not the best approach. There are at least two drawbacks to it. Firstly, the user may want to have the option to overwrite the unit price. Secondly, if the price of the product changes the amounts for all your invoices changes too, this is not good.

To avoid these drawbacks it's better to store the price of the product for each detail. Let's add a `pricePerUnit` persistent property to the `Detail` entity and let's calculate its value from the `price` in `Product` using a `@DefaultValueCalculator`. Just to obtain the effect you can see in figure 5.2.



Detail

Product Nº	1	Product	Peopleware: Productive Projects and Teams
Quantity	2	Price per unit	31.00 €
		Amount	62.00 €

Save detail Close Remove detail

When the user chooses a product...
... the price per unit gets value from product price. The user can overwrite it
When quantity or price per unit change amount is recalculated

Figure 5.2 Detail entity with pricePerUnit as persistent property

The first obvious step is to add the property `pricePerUnit`. Add the code in listing 5.5 to your `Detail` entity.

Listing 5.5 The pricePerUnit property of Detail entity

```
@DefaultValueCalculator(
    value=PricePerUnitCalculator.class,  // This class calculates the initial value
    properties=@PropertyValue(
       name="productNumber",  // The productNumber property of the calculator...
       from="product.number") // ...is filled from product.number of the entity
)
@Stereotype("MONEY")
private BigDecimal pricePerUnit;  // A regular persistent property...

public BigDecimal getPricePerUnit() {  // ...with its getter and setter
    return pricePerUnit==null?
       BigDecimal.ZERO:pricePerUnit;  // Thus never returns null
}

public void setPricePerUnit(BigDecimal pricePerUnit) {
    this.pricePerUnit = pricePerUnit;
}
```

`PricePerUnitCalculator` contains the logic to calculate the initial value. It simply reads the price from the product. See the code for this calculator in listing 5.6.

Listing 5.6 PricePerUnitCalculator calculates the default value for pricePerUnit

```
package org.openxava.invoicing.calculators;  // In 'calculators' package

import org.openxava.calculators.*;
import org.openxava.invoicing.model.*;

import static org.openxava.jpa.XPersistence.*;  // For using getManager()

public class PricePerUnitCalculator implements ICalculator {

    private int productNumber;  // Contains the product number when calculate() is called

    public Object calculate() throws Exception {
       Product product = getManager()  // getManager() from XPersistence
          .find(Product.class, productNumber);  // Find the product
```

```
        return product.getPrice();  // Returns its price
    }

    public void setProductNumber(int productNumber) {
        this.productNumber = productNumber;
    }

    public int getProductNumber() {
        return productNumber;
    }

}
```

In this way when the user chooses a product the price per unit field is filled with the price of that product but because it's a persistent property, the user can change it. And if in the future the price of the product changes this price per unit of the detail will not change.

This means that you have to adapt your `amount` calculated property (listing 5.7).

Listing 5.7 The amount property of Detail adapted to use pricePerUnit

```
@Stereotype("MONEY")
@Depends("pricePerUnit, quantity")  // pricePerUnit instead of product.number
public BigDecimal getAmount() {
    return new BigDecimal(quantity)
        .multiply(getPricePerUnit());  // getPricePerUnit() instead  of product.getPrice()
}
```

The `getAmount()` method uses `pricePerUnit` as source instead of `product.price`.

Finally, we have to modify the `Detail` view to show the new property (listing 5.8).

Listing 5.8 View of Product now includes pricePerUnit

```
@View(members="product; quantity, pricePerUnit, amount")  // pricePerUnit added
public class Detail extends Identifiable {
```

And the `Simple` view for `Product` to remove the display of the price (listing 5.9).

Listing 5.9 Simple view of Product now does not include price

```
@View(name="Simple", members="number, description")  // price removed
public class Product {
```

That is, we have arranged the user interface to show the persistent (and modifiable by the user) `pricePerUnit` property instead of the price from the product.

Also it would be nice to see in the collection these new properties. To do so, edit the `CommercialDocument` entity and modify the list of properties to show in the collection, just as shown in listing 5.10.

Listing 5.10 Adding pricerPerUnit and amount to details in ComercialDocument

```
@ListProperties(
    "product.number, product.description, " +
    "quantity, pricePerUnit, amount")  // pricePerUnit and amount added
private Collection<Detail> details = new ArrayList<Detail>();
```

Update your database schema, try the `Order` and `Invoice` modules and observe the new behavior when adding details.

5.1.3 Calculated properties depending on a collection

We want to add amounts to `Order` and `Invoice` too. To calculate vat, base amount and total amount are indispensable. To do so you only need to add a few calculated properties. Figure 5.3 shows the user interface for these properties.

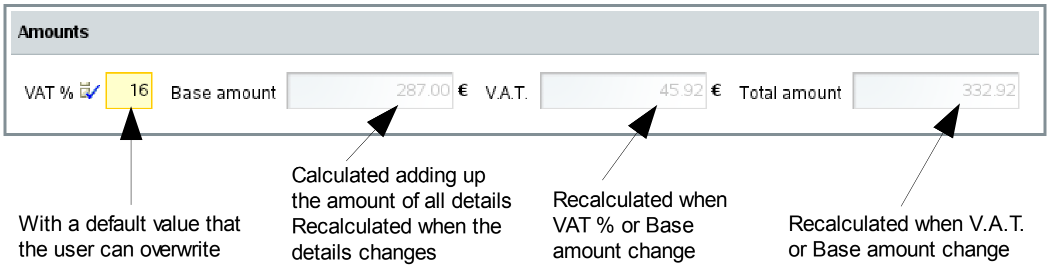

Amounts
VAT %
16
Base amount
287.00 €
V.A.T.
45.92 €
Total amount
332.92
With a default value that the user can overwrite
Calculated adding up the amount of all details Recalculated when the details changes
Recalculated when VAT % or Base amount change
Recalculated when V.A.T. or Base amount change

Figure 5.3 Calculated properties for CommercialDocument

Let's start with `baseAmount`. Listing 5.11 shows its implementation.

Listing 5.11 Calculated baseAmount property in CommercialDocument

```
@Stereotype("MONEY")
public BigDecimal getBaseAmount() {
    BigDecimal result = new BigDecimal("0.00");
    for (Detail detail: getDetails()) {  // We iterate through all details
       result = result.add(detail.getAmount());   // Adding up the amount
    }
    return result;
}
```

The implementation is simple, just adding up the amount from all the details.

The next property to add is `vatPercentage` to calculate the vat. See listing 5.12.

Listing 5.12 The persistent vatPercentage property in CommercialDocument

```
@Digits(integerDigits=2, fractionalDigits=0)  // To indicate its size
@Required
private BigDecimal vatPercentage;
```

```
public BigDecimal getVatPercentage() {
    return vatPercentage==null?
       BigDecimal.ZERO:vatPercentage;   // Thus never returns null
}

public void setVatPercentage(BigDecimal vatPercentage) {
    this.vatPercentage = vatPercentage;
}
```

You can see that `vatPercentage` is a conventional persistent property. In this case we use `@Digits` (an annotation from the Hibernate Validator framework) as an alternative to `@Column` to specify the size.

We continue adding the `vat` property. See it in listing 5.13.

Listing 5.13 The calculated vat property in CommercialDocument

```
@Stereotype("MONEY")
@Depends("vatPercentage")  // When vatPercentage changes vat is
public BigDecimal getVat() {   // recalculated and redisplayed
    return getBaseAmount()   // baseAmount * vatPercentage / 100
       .multiply(getVatPercentage())
       .divide(new BigDecimal("100"));
}
```

It's a simple calculation. We use `@Depends` in order to recalculate and redisplay `vat` each time the user modifies `vatPercentage`.

Only `totalAmount` remains. You can see its code in listing 5.14.

Listing 5.14 The calculated totalAmount property in CommercialDocument

```
@Stereotype("MONEY")
@Depends("baseAmount, vat")  // When baseAmount or vat changes totalAmount is
public BigDecimal getTotalAmount() {  // recalculated and redisplayed
    return getBaseAmount().add(getVat());   // baseAmount + vat
}
```

Again a simple calculation and again we use `@Depends`.

Now that you have written the amount properties of `CommercialDocument`, you must modify the view in order to show these new properties (listing 5.15).

Listing 5.15 Adding amount properties to the view for CommercialDocument

```
@View(members=
    "year, number, date;" +
    "data {" +
       "customer;" +
       "details;" +
       "amounts [ " +  // Square brackets means a group, displayed inside a frame
       "  vatPercentage, baseAmount, vat, totalAmount" +
       "];" +
       "remarks" +
    "}"
)
abstract public class CommercialDocument extends Identifiable {
```

We add vatPercentage, baseAmount, vat and totalAmount between details and remarks but inside square a brackets with the name (amounts). This means that these properties will be shown inside a frame named “amounts”. This is a group.

Now you can update the database schema (because of vatPercentage) and try your application. It would behave almost as in the aforesaid figure 5.3. “Almost” because vatPercentage does not have a default value yet. We add it in the next section.

5.1.4 Default value from a properties file

It's useful for the user to have the default value populated for the vatPercentage. You can use calculator (with @DefaultValueCalculator) that returns a fixed value. In this case changing the default value means changing your source code. Otherwise you can read the default value from the database (using JPA from your calculator). In that case changing the default value means updating a database table.

Another option is to store this configuration value in a properties file, a plain file with key=value pairs. In this case changing the default value for vatPercentage is just a matter of editing a plain file with a text editor.

Let's implement the properties file option. Create a file named *invoicing.properties* in the *Invoicing/properties* folder with the content of listing 5.16.

Listing 5.16 The content of invoicing.properties file

```
defaultVatPercentage=21
```

Though you can use the java.util.Properties class from Java to read this file we prefer to create a custom class to read these properties. We are going to call this class InvoicingPreferences and we'll put it in a new package named org.openxava.invoicing.util. You have the code in listing 5.17.

Listing 5.17 InvoicingPreferences to read invoicing.properties file

```
package org.openxava.invoicing.util;   // In 'util' package

import java.io.*;
import java.math.*;
import java.util.*;

import org.apache.commons.logging.*;
import org.openxava.util.*;

public class InvoicingPreferences {
    private final static String
       FILE_PROPERTIES="invoicing.properties";
    private static Log log =
       LogFactory.getLog(InvoicingPreferences.class);
    private static Properties properties;   // We store the properties here

    private static Properties getProperties() {
       if (properties == null) {  // We use lazy initialization
          PropertiesReader reader =  // PropertiesReader is a utility class
             new PropertiesReader(   // from OpenXava
                InvoicingPreferences.class,
                FILE_PROPERTIES);
          try {
             properties = reader.get();
          }
          catch (IOException ex) {
          log.error(
             XavaResources.getString(  // To read a i18n message
                "properties_file_error",
                FILE_PROPERTIES),
             ex);
             properties = new Properties();
          }
       }
       return properties;
    }


    public static BigDecimal getDefaultVatPercentage() {  // The only public method
        return new BigDecimal(
       getProperties().getProperty("defaultVatPercentage"));
    }

}
```

As you can see `InvoicingPreferences` is a class with one static method, `getDefaultVatPercentage()`. The advantage of this class approach over the properties files is that if you change the way the preferences are obtained, for example reading from a database or an LDAP directory, you only have to change this class in your entire application.

You can use this class from the default calculator for the `vatPercentage` property. See the calculator in listing 5.18.

Listing 5.18 Calculator for default value of vat percentage

```
package org.openxava.invoicing.calculators;  // In 'calculators' package

import org.openxava.calculators.*;  // To use ICalculator
import org.openxava.invoicing.util.*;  // To use InvoicingPreferences

public class VatPercentageCalculator implements ICalculator {

    public Object calculate() throws Exception {
       return InvoicingPreferences.getDefaultVatPercentage();
    }
}
```

As you see, it just returns the `defaultVatPercentage` from `InvoicingPreferences`. Now, you can use this calculator in the definition of `vatPercentage` property in `CommercialDocument`. See listing 5.19.

Listing 5.19 Default calculator for vatPercentage of CommercialDocument

```
@DefaultValueCalculator(VatPercentageCalculator.class)
private BigDecimal vatPercentage;
```

With this code when the user clicks to create a new invoice, the `vatPercentage` field will be filled with 21 or whatever other value you put in *invoicing.properties*.

5.2 JPA callback methods

Another useful way to add business logic to your model is using JPA callback methods. A callback method is a method in your entity that is called in some specific moment of its life cycle as a persistent object. That is, you can specify some logic to execute on save, read, remove or modification of the entity.

In this section we'll see some practical applications of JPA callback methods.

5.2.1 Multiuser safe default value calculation

Until now we were calculating the `Invoice` and `Order` number using `@DefaultValueCalculator`. This calculates the default value when the user clicks to create a new `Invoice` or `Order`. So, if several users click on the “new” button at the same time all of them get the same number. This is not multiuser safe. The way to generate a unique number is by generating it just on save.

We are going to implement it using a JPA callback method. JPA allows you to mark any method of your class to be executed in any part of its life cycle. We'll indicate the calculation of the number just before the saving of the `CommercialDocument`. Using this approach we'll improve the number calculation for having a different numeration for `Order` and `Invoice`.

Edit the `CommercialDocument` entity and add the `calculateNumber()` method you have in listing 5.20.

Listing 5.20 The @PrePersist method calculateNumber() of CommercialDocument

```
@PrePersist  // Executed just before saving the object for the first time
public void calculateNumber() throws Exception {
    Query query = XPersistence.getManager()
       .createQuery("select max(i.number) from " +
            getClass().getSimpleName() +  // Thus it's valid for both Invoice and Order
            " i where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    this.number = lastNumber == null?1:lastNumber + 1;
}
```

The code in listing 5.20 is the same as that of the NextNumberForYearCalculator but using getClass().getSimpleName() instead of "CommercialDocument". The getSimpleName() method returns the name of the class without the package, i.e., just the entity name. It will be "Order" for Order and "Invoice" for Invoice. Thus we can get a different numeration for Order and Invoice.

JPA specification states that you should not use JPA API inside a JPA callback method. So the above method is not legal from a strict JPA viewpoint. But, Hibernate (the JPA implementation OX uses by default) allows you to use it in @PrePersist. And since JPA is the easier way to do this calculation we use it in our practice.

Now you can delete the NextNumberForYearCalculator class from your project, and modify the number property of CommercialDocument to avoid using it (listing 5.21).

Listing 5.21 number of CommercialDocument without @DefaultValueCalculator

```
@Column(length=6)
~~@DefaultValueCalculator(value=NextNumberForYearCalculator.class,~~ // Remove this
    ~~properties=@PropertyValue(name="year")~~
~~)~~
@ReadOnly  // The user cannot modify the value
private int number;
```

Note that in addition to removing @DefaultValueCalculator, we have added the @ReadOnly annotation. This means that the user cannot enter or modify the number. This is the right approach given that the number is generated on saving the object, so the user typed value would always be overridden.

Try now the Invoice or Order module and you will see that the number is empty and not editable, and when you add the first detail, that saves the container document, the document number is calculated and updated in the user interface.

5.2.2 *Synchronizing persistent and calculated properties*

The approach towards calculating the vat, base amount and total amount of a commercial document is natural and practical. We used calculated properties that calculate, using pure Java, the values each time they are called. This is good because when the user adds, modifies or removes a detail in the user interface, the vat, base amount and total amount values are recalculated with fresh data instantly.

But, calculated properties have some drawbacks. For example, if you want to do batch processing or a report for all invoices with a total amount between some range, it cannot be done using calculated properties. If you have a huge database the process will be very slow, because you will have to instantiate all invoices for calculating its total amount. For these cases having a persistent property, a column in the database, for the invoice or order amount, can yield far better performance.

In our case we'll maintain the current calculated properties, but we are going to add a new one, called `amount`, that will contain the same value as `totalAmount`, but this `amount` will be persistent with a corresponding column in the database. The tricky issue here is to have the `amount` property synchronized. We are going to use the callback JPA method in order to achieve this.

The first step is to add the `amount` property to `CommercialDocument`. Just plain vanilla, see it in listing 5.22.

Listing 5.22 Persistent property amount in CommercialDocument

```
@Stereotype("MONEY")
private BigDecimal amount;

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}
```

Let's modify the `Detail` entity so that every time a detail is added, removed or modified the `amount` property of its container `CommercialDocument` will be correctly recalculated. Just add the three callback methods of listing 5.23 to your `Detail` entity.

Listing 5.23 Callback methods in Detail to recalculate parent amount

```
@PrePersist   // On save the detail the first time
private void onPersist() {
    getParent().getDetails().add(this);   // To have the collection synchronized
    getParent().recalculateAmount();
}

@PreUpdate   // Every time the detail is modified
private void onUpdate() {
    getParent().recalculateAmount();
}

@PreRemove   // On remove the detail
```

```
private void onRemove() {
    getParent().getDetails().remove(this);  // To have the collection synchronized
    getParent().recalculateAmount();
}
```

Basically, we call the `recalculateAmount()` method of the container `CommercialDocument` each time a detail is added, removed or modified. Let's see this method in listing 5.24.

Listing 5.24 recalculateAmount() synchronizes totalAmount and amount

```
public void recalculateAmount() {
    setAmount(getTotalAmount());
}
```

As you can see we moved the calculated property, `totalAmount`, to the persistent property, `amount`.

You can try the `Invoice` or `Order` module with this code, and you will see how when a detail line is added, removed or modified, the column in the database for `amount` is correctly updated. However, if you try to remove the `Invoice` or `Order` you get an exception (`java.util.ConcurrentModificationException`). This is because we're using cascade ALL, and when a `CommercialDocument` is removed its details are automatically removed, and in this case the `@PreRemove` callback method (`onRemove()`) fails. We need to do a little workaround to avoid this effect. The workaround is simple: just do not execute the `onRemove()` method of `Detail` when the container `CommercialDocument` is being removed. To program it add the code in listing 5.25 to your `CommercialDocument` class.

Listing 5.25 Code in CommercialDocument to indicate that it is removing

```
@Transient   // It's not stored in database table
private boolean removing = false;     // Indicates if JPA is removing the
                                      // CommercialDocument now
boolean isRemoving() {  // Package access, it cannot be accessed from outside
    return removing;
}

@PreRemove   // When the document starts to be removed we set removing as true
private void markRemoving() {
    this.removing = true;
}

@PostRemove   // When the document has been removed we set removing as false
private void unmarkRemoving() {
    this.removing = false;
}
```

You see how we have added a transient property `removing`, which is true only when the `CommercialDocument` is being removed. We can use this property from

`Detail` in the way you see in listing 5.26.

Listing 5.26 Refined onRemoved() method of Detail

```
@PreRemove
private void onRemove() {
    if (getParent().isRemoving()) return;  // We add this line to avoid exceptions
    getParent().getDetails().remove(this);
    getParent().recalculateAmount();
}
```

That is, if the container is being removed, we do not execute the logic for `onRemove()`. After this little refinement, `Invoice` and `Order` have its `amount` property always synchronized, and ready to be used in massive processing.

5.3 Database logic (@Formula)

Ideally you write all your business logic in Java, inside your entities. Nevertheless, sometimes that is not the most convenient way. Imagine that you have a calculated property in `CommercialDocument`, let's say `estimatedProfit`, like the one in listing 5.27.

Listing 5.27 estimatedProfit as a calculated property in CommercialDocument

```
@Stereotype("MONEY")
public BigDecimal getEstimatedProfit() {
    return getAmount().multiply(new BigDecimal("0.10"));
}
```

If you need to process all those invoices with an `estimatedProfit` greater than 1000, you have to code something like listing 5.28.

Listing 5.28 Selecting objects depending on a calculated property

```
Query query = getManager()
    .createQuery("from Invoice");  // No condition in query
for (Object o:  query.getResultList()) {  // Iterates over all objects
    Invoice i = (Invoice) o;
    if (i.getEstimatedProfit()  // Queries every object
       .compareTo(new BigDecimal("1000")) > 0) {
       i.doSomething();
    }
}
```

You cannot use a condition in the query to discriminate by `estimatedProfit`, because `estimatedProfit` is not in the database, it's only in the Java object, so you have to instantiate every object in order to ask by the estimated profit. In some cases this way is a good option, but if you have a really huge amount of invoices, and only a few of them have the `estimatedProfit` greater than 1000, then your process will be very inefficient. What alternative do we have?

Our alternative is to use the `@Formula` annotation. `@Formula` is a Hibernate extension to the JPA standard, that allows you to map a property to a SQL statement. You can define `estimatedProfit` with `@Formula` as shown in listing 5.29.

Listing 5.29 estimatedProfit as a @Formula property in CommercialDocument

```
@org.hibernate.annotations.Formula("AMOUNT * 0.10")  // The calculation using SQL
@Stereotype("MONEY")
private BigDecimal estimatedProfit;  // A field, as in the persistent property case

public BigDecimal getEstimatedProfit() {  // Only the getter is needed
    return estimatedProfit;
}
```

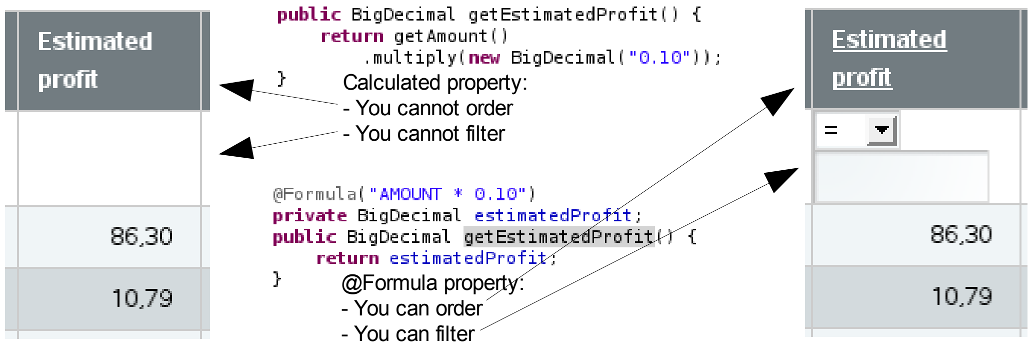
This means that when a `CommercialDocument` is read from the database, the `estimatedProfit` field will be filled with the calculation for `@Formula` that is done by the database. The most useful thing of `@Formula` properties is that you can use it in different conditions, so that you can rewrite the above process as shown in listing 5.30.

Listing 5.30 Selecting objects depending on a @Formula property

```
Query query = getManager()
    .createQuery("from Invoice i where " +
       "i.estimatedProfit > :estimatedProfit");  // Condition allowed
query.setParameter("estimatedProfit", new BigDecimal(1000));
for (Object o:  query.getResultList()) {  // Iterates only over selected objects
    Invoice i = (Invoice) o;
    i.doSomething();
}
```

In this way we put the weight of calculating the estimated profit and selecting the record on the database server, and not on the Java server.

This fact also has effect in the list mode, because the user cannot filter or order by calculated properties, but he can do so using `@Formula` properties (figure 5.4)



```
public BigDecimal getEstimatedProfit() {
    return getAmount()
        .multiply(new BigDecimal("0.10"));
}
```

Calculated property:
- You cannot order
- You cannot filter

```
@Formula("AMOUNT * 0.10")
private BigDecimal estimatedProfit;
public BigDecimal getEstimatedProfit() {
    return estimatedProfit;
}
```

@Formula property:
- You can order
- You can filter

Figure 5.4 List mode for calculated and @Formula properties

`@Formula` is a good option for improving performance in some cases. Anyways, generally it's better to use calculated properties and writing your logic in Java. The advantage of calculated properties over `@Formula` is that your code is not database dependent. Moreover with calculated properties you can re-execute the calculation without reading the object from the database, so that you can use `@Depends`.

5.4 JUnit tests

Before we move on to the next lesson, we are going to write the JUnit code for this one. Remember, the code is not done if it has no tests. You can write the tests before, during or after coding. But you always have to write the tests.

The test code here is not only to give you a good example, but also to teach you ways to test different cases in your OpenXava application.

5.4.1 Modifying existing tests

Creating a new test for each new case seems like a good idea from a structural viewpoint, but in most cases it is not practical because in doing so your test code would grow very fast, and execution of all the tests for your application would take a substantial amount of time.

The more pragmatic approach is to modify the existing test code to cover all the new cases we have developed. Let's do it in this way.

In our case, all the code for this lesson applies to `CommercialDocument`, so we are going to modify the `testCreate()` method of `CommercialDocumentTest` to match the new functionality. We leave the `testCreate()` method as you see in listing 5.31.

Listing 5.31 The modified testCreate() method of CommercialDocumentTest

```
public void testCreate() throws Exception {
    calculateNumber();   // Added to calculate the next document number first
    verifyDefaultValues();
    chooseCustomer();
    addDetails();
    setOtherProperties();
    save();
    verifyAmountAndEstimatedProfit();   // To test callback  method and @Formula
    verifyCreated();
    remove();
}
```

As you see, we add a new line in the beginning to calculate the next document number, and call the new `verifyAmountAndEstimatedProfit()` method.

Now it's more convenient to calculate the next document number in the beginning to use it during the test. To do this, change the old `getNumber()` method for the two methods in listing 5.32.

Listing 5.32 A method to calculate the next number and another to get it

```
private void calculateNumber() {
    Query query = getManager().
       createQuery(
          "select max(i.number) from " +
          model +  // We change CommercialDocument for a variable
          " i where i.year = :year");
    query.setParameter("year", Dates.getYear(new Date()));
    Integer lastNumber = (Integer) query.getSingleResult();
    if (lastNumber == null) lastNumber = 0;
    number = Integer.toString(lastNumber + 1);
}

private String getNumber() {
    return number;
}
```

Previously we only had `getNumber()` to calculate and return the number, now we have a method to calculate (`calculateNumber()`), and another one to return the result (`getNumber()`). You can note that the calculation logic has a little change, instead of using "CommercialDocument" as the source of the query we use `model`, a variable. This is because now the numeration for invoices and orders are separated. We fill this variable, a field of the test class, in the test constructor, just as shows in listing 5.33.

Listing 5.33 CommercialDocumentTest constructor stores the model name

```
private String model;   // The model name to use in the query. Can be "Invoice" or "Order"

public CommercialDocumentTest(String testName, String moduleName) {
    super(testName, "Invoicing", moduleName);
    this.model = moduleName;   // In this case module name matches model
}
```

In this case module name, Invoice or Order, coincides with model name, Invoice or Order, so the easiest way to get the model name is from the module name.

Let's see the actual testing of the new functionality.

5.4.2 Testing default values and calculated properties

In this lesson we have done some modifications related to default values. Firstly, the default value for `number` is not calculated by means of `@DefaultValueCalculator` instead we use a JPA callback method. Secondly, we have a new property, `vatPercentage`, whose initial value is calculated by reading from a property file. To test these cases we have to modify the `verifyDefaultValues()` method as you see in listing 5.34.

Listing 5.34 The verifyDefaultValues() method adapted

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertValue("year", getCurrentYear());
    ~~assertValue("number", getNumber());~~  // Now number has no initial value
    assertValue("number", "");   // on create a new document (section 5.2.1)
    assertValue("date", getCurrentDate());
    assertValue("vatPercentage", "21");// Default value from properties file (section 5.1.4)
}
```

We test the `vatPercentage` default value calculation and we verify that the `number` has no initial value, because now the `number` is not calculated until the document is saved (section 5.2.1). When the document (invoice or order) will be saved we'll verify that the `number` is calculated. The document is saved just when the first detail is added. Also when the detail is added we can test the `amount` for `Detail` calculation (the simple calculated property, section 5.1.1), the default value calculation for `pricePerUnit` (`@DefaultValueCalculator`, section 5.1.2) and the amount properties of the document (calculated properties depending on a collection, section 5.1.3). We'll test all this with a few modifications in the already existing `addDetails()` method (listing 5.35).

Listing 5.35 addDetails() now tests calculated properties and default values

```
private void addDetails() throws Exception {
    // Adding a detail line
    assertCollectionRowCount("details", 0);
    execute("Collection.new",
       "viewObject=xava_view_section0_details");
    setValue("product.number", "1");
    assertValue("product.description",
       "Peopleware: Productive Projects and Teams");
    assertValue("pricePerUnit",   // @DefaultValueCalculator, section 5.1.2
       "31.00");
    setValue("quantity", "2");
    assertValue("amount", "62.00");   // Calculated property, section 5.1.1
    execute("Collection.save");
    assertNoErrors();
    assertCollectionRowCount("details", 1);

    // On saving first detail the document is saved,
    // then we verify that number is calculated
    assertValue("number", getNumber());  // Multiuser safe default value, section 5.2.1

    // Verifying calculated properties of document
    assertValue("baseAmount", "62.00");    // Calculated properties
    assertValue("vat", "11.16");                // depending on a collection,
    assertValue("totalAmount", "73.16");   // section 5.1.3

    // Adding another detail
    execute("Collection.new",
       "viewObject=xava_view_section0_details");
    setValue("product.number", "2");
    assertValue("product.description",
       "Arco iris de lágrimas");
    assertValue("pricePerUnit",   // @DefaultValueCalculator, section 5.1.2
          "15.00");
      setValue("pricePerUnit", "10.00");  // Modifying the default value
      setValue("quantity", "1");
      assertValue("amount", "10.00");   // Calculated property, section 5.1.1
      execute("Collection.save");
      assertNoErrors();
      assertCollectionRowCount("details", 2);

      // Verifying calculated properties of document
      assertValue("baseAmount", "72.00");    // Calculated properties
      assertValue("vat", "12.96");            // depending on a collection,
      assertValue("totalAmount", "84.96");   // section 5.1.3
  }
```

As you see, with these simple modifications we test most of our new code. What remains is only the `amount` and `estimatedProfit` properties. We'll test them in the next section.

5.4.3 Testing calculated and persistent synchronized properties / @Formula

In section 5.2.2 we used a JPA callback method in `CommercialDocument` to have a persistent property, `amount`, synchronized with a calculated one, `totalAmount`. The `amount` property is only shown in list mode.

In section 5.3 we have created a property that uses `@Formula`, `estimatedProfit`. This property is shown only in list mode.

Obviously, the simplest way to test it is by going to list mode and verifying that the values for these two properties are the expected ones. You have already seen that in `testCreate()` we call the `verifyAmountAndEstimatedProfit()`. Let's see its code in listing 5.36.

Listing 5.36 The new method verifyAmountAndEstimatedProfit()

```
  private void verifyAmountAndEstimatedProfit() throws Exception {
      execute("Mode.list");  // Changes to list mode
      setConditionValues(new String [] { // Filters to see in the list
          getCurrentYear(), getNumber()   // only the newly created document
      });
      execute("List.filter");  // Does the filter
      assertValueInList(0, 0, getCurrentYear());  // Verifies that
      assertValueInList(0, 1, getNumber());  // the filter has worked
      assertValueInList(0, "amount", "84.96");  // Asserts amount
      assertValueInList(0, "estimatedProfit", "8.50");  // Asserts estimatedProfit
      execute("Mode.detailAndFirst");  // Goes to detail mode
  }
```

Because we now go to list mode and then we go back to detail. We have to make a small modification to the `verifyCreated()` method, that is executed just after `verifyAmountAndEstimatedProfit()`. Let's see the modification in listing 5.37.

Listing 5.37 verifyCreated() modified to not search the newly created document

```
private void verifyCreated() throws Exception {
    ~~setValue("year", getCurrentYear());~~   // We delete these lines
    ~~setValue("number", getNumber());~~   // for searching the document
    ~~execute("CRUD.search");~~   // because we already searched it with list mode

    // The rest of the test...
    ...
```

We remove these lines because now it's not necessary to search the newly created document. Now in the `verifyAmountAndEstimatedProfit()` method we went to list mode and chose the document, so we are already editing the document.

Congratulations! Now you have your tests up to date with your code. It's a good time to run all the tests for your application.

5.5 Summary

In this lesson you have learned some common ways to add business logic to your entities. There should be no doubt about the utility of calculated properties, callback methods or `@Formula`. Nevertheless, there are many other ways to add logic to your OpenXava application, and we are going to learn them.

In the coming lessons you'll see how to add validation, modify the standard module behavior and add your own business logic, among other ways to add custom logic to your application.