## Chapter 6: Creating a Domain Model

Any developer who has spent any time developing enterprise applications will agree that the hardest part of programming is to understand the Customers Business i.e. The domain of the application. It therefore stands to reason that you should build your software to match the domain of the business as closely as possible hence the domain model. Resulting in a single shared language between the programmer, the program and the businessPossibly out of place here. The domaim model is independent of the UI.

Although this is not a book on Software development methodologies while training developers on Domain Modelling we are often asked how you go about building a domain model, using firestarter and developing a system using Habanero. We have found that there is no one way of doing things and that each project has a different ideal approach depending on the environment e.g. The knowledge of the customer on what he/she wants, the experience of the team in the domain. the experience of the team with the technologies, the physical, business and emotional relationship with the customer, the experience that the team has working together, the inherint risk of the project (i.e. an aircraft control system is higher risk than a library admin system).

These are difficult to quantify but we have found that the higher the risk the shorter the iterations are between feedback.

If you are replacing an existing system and making virtually no changes to the functionality of the system, you understand the technologies well and have reasonable experience in the domain then you can probably create a fairly complete domain model up front and implement this domain model. Your model would typically contain at least a class diagram and if you are using a relational database then an ERD. You can implement the database design and reverse engineer the classes from the database design using Firestarters powerfull reverse engineering capabilities. The Businesss Objects, relationships, properties and Object to relational mappings will all be modelled for you where the firestarter wizards will assist you with these mappings.

As you develop the system you may find minor changes in which case you can modify the model in Firestarter and regenerate the code. During maintenance when additional requirments are found the model can be updated in Firestarter and the code regenerated. Firestarter also generates basic (CRUD) user interfaces but the user of Firestarter for this will be discussed in section

If you are developing a system for a business application that you do not know well or a new type of application where the customer does not even really know what they want then you should create a detailed model of only the smallest piece possible (System under development). You should iterate between Analysing Modelling, Developing and testing with the customer Stakeholders (user/customer/business analyst) as quickly as possible. In addition you should only detail the domain model when needed. Following the agile principle of delaying detailed design until needed.

Firestarter has been designed to be used in the most agile manner possible and in fact we have found that using Firestarter to manage the refactoring of the Domain model has a massive impact on the agility of a project and on the management of change in the domain model.

The best way to demonstrate Firestarter being used in this manner to to work through an example.

DO A USE CASE

Implement test 1

Model a business object required

Generate

Expand test 1

Add to model

Generate

……

Note: At this point you need not have developed any database since all the tests are executing against a memory database for agile development this creates an incredible benefit since the software can be developed, unit tested prototyped and tested for usability and usefullness without having to manage a relational database. This significantly increases the agility of the process since the model can be refactored in Firestarter with ease.

If you have a class model in place a good start is to do a one-to-one mapping of your classes to data tables, an approach that works well in "greenfield" environments where you have the luxury of designing your database schema from scratch. Because this rarely happens in practice you need to be prepared to be constrained by one or more legacy database schemas. which you will then need to map your classes to. In these situations it is unlikely that you will need to do much data modeling; you will simply need to learn to live with the existing data sources, but you will need to be able to read and understand existing models. In some cases you may need to perform legacy data analysis ([Chapter 14](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0138.html#975)) and model the existing schema before you can start working with it. From Scott Ambler

Refer to Doman-Driven Design – Tackling complexity in the heart of software by Eric Evans.

A rich domain model on a large project provides an ubiquitous language for the entire team to communicate in the language of the problem domain. A good domain model with the appropriate automated tests provides the basis for software that can continue to grow and improve with the business instead of suffering from ‘software entropy’ and becoming legacy software. A domain model is a model of reality and will thus only represent the aspects of reality that are relevant to the particular problem being solved by the project. The domain of a software program is the subject area of interest and activities that the software is used. To be able to create a useful software program it is essential that the team acquires the required domain knowledge. A domain model can be expressed as diagrams, text or code, the domain model does not have to match ‘reality’ but is instead a model of the characteristics of the domain that are relevant for the particular set of problems and/or activities being solved by the software program.

The utility of a domain model – from evans pg 4

A domain model and the software design and implementation shape each other and map to each other. The domain model is represented throughout the actual software and will be useful during software maintenance.

The domain model provides a common language that is used by all team members. This provides the developers with a common language that they can talk about their programs with as well as with which to communicate with domain experts.

The model provides distilled knowledge and as such distinguishes the areas of most interest. The user of a rich model through all the versions of the system allows the various versions of the system to feedback to the domain model allowing the model to grow and develop.

Evans Emphasises repeatedly in his book that the Domain model is not a set of diagrams a requirement/design document but is instead a language which becomes ubiquitous and is present in the diagrams the actual code, the tests, the coffee machine chats and the conversations held between developers different modules, analysts, domain experts and users. With Model-Driven design the model is owned by all members of the team. The old divide of Analysts, Designers and programmers who communicate primarily via UML and documentation does not work. The modeller must be involved to some extent in the design and the programmer is always involved in the model.

*What gets modelled in this*? The business objects of interest to the business, the relationships to them, the rules for updating editing and deleting business objects as well as security relating to using the business objects.

Examples

* An *Invoice* has one or more *Invoice Lines*.
* If an *Invoice* is deleted then all its *Invoice lines* must be deleted.
* An *Invoice* that has been revenue recognised can never be deleted.
* An *Invoice* is created for one and only one customer order. The *Customer Order* must be approved for an invoice to be created.
* Once *Invoice line* is created for each *customer order item* that the *Invoice* is associated with.
* An *Invoice line* will be associated with one or more *Case*. Where a *Case* is a single package of a specific *Product* delivered to a *Customer*.
* An *Invoice Line Value* is compulsory and must be greater than zero.
* An *Invoice line* must always be associated with an individual *Invoice*. Each *Invoice line* has a line number. An invoice cannot have two invoice lines with the same invoice line number.
* Only users of the ‘Account Manager’ profile can add or remove cases from an invoice. Cases can never be added or removed if the Invoice has been revenue recognised.
* Only users of the ‘Customer Service’ Profile can create an invoice.
* Only users of the ‘Accounting’ Profile can revenue recognise an invoice.

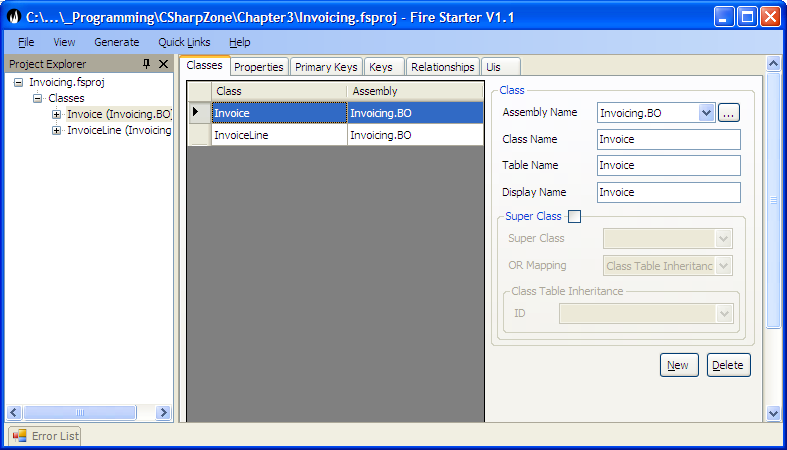
There are many more rules for this system but these have been selected as examples of rules that are modelled in the domain layer. In addition the system has been separated into separate sub systems as per this diagram. For the purposes of this example the Customer and all related rules and data have been left out of the model.

Capturing the Domain Model using Firestarter

Firestarter (a Domain modeller – program generator and Data Mapper) and Habanero (an application framework) work together to assist the application developer to focus on and implement a domain model like this without having to focus on the complexities of Object-Relational mapping, Security implementation, Business rule implementation etc. This frees the application developer up to concentrate on the modelling and developing the core business problems. This not only results in dramatic improvements in productivity but also in the quality of the final application.

The first thing we do is model the two Business Objects modelled as part of the Invoicing sub system in the above analysis.

The XML file containing the definition discussed here is contained in downloads – Chapter3\Invoicing.fsproj.

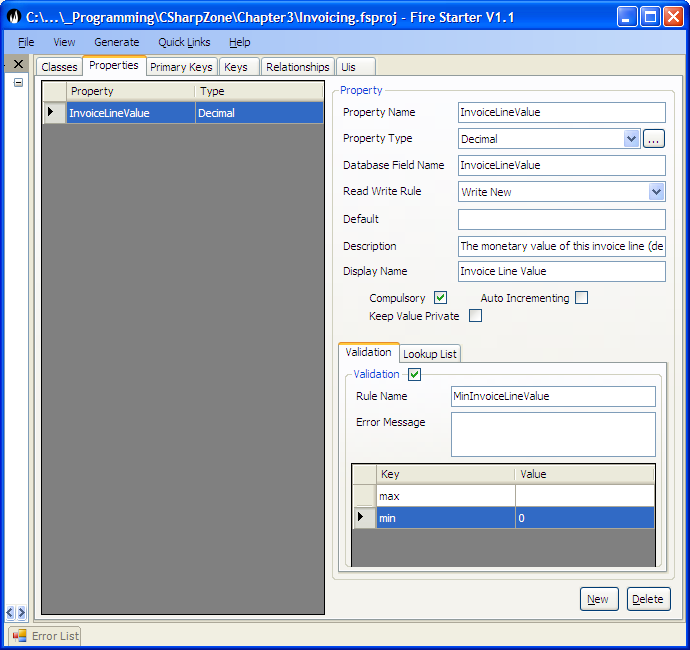


Then we model the various properties that each Business Object has in our case the Invoice Line has a line number – the sequence number of the line printed on the invoice, a line description – a textual description of the invoice line and a value the amount that the customer is being charged.

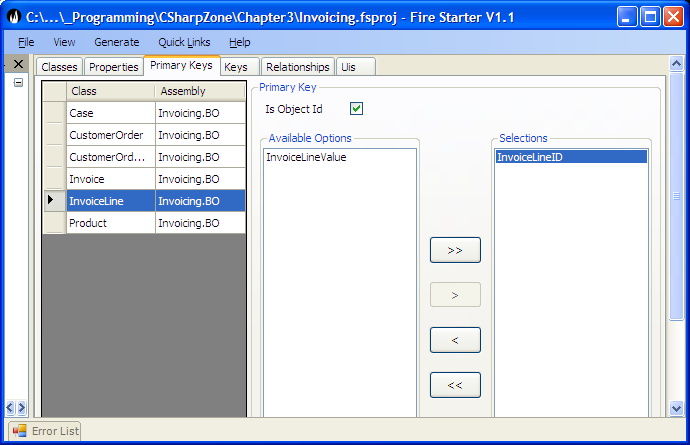
Each of these properties of a business object can have rules set for it. We will demonstrate this by adding the property *Invoice Line Value* to *Invoice Line*.

From this we can see that the *InvoiceLineValue* is

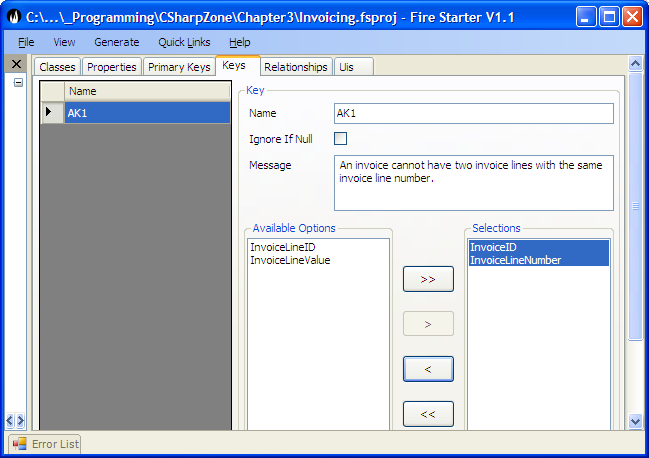
* Type Decimal.
* Read Write Rule of Write New i.e. it must be set to a valid value before the object can be persisted for the first time else the object will be in an invalid state. The *InvoiceLineValue* can never be updated after the object has been persisted to the database. The other options for Read Write Rule are
* Default value. If the property has any default value that is set when a new Business object is created.
* Description: This is the description of a business object property which is used for tool tip text on user interface, generated XML documentation for the property etc.
* Compulsory: If true the business object will not be in a valid state and will not be persistable to a datasource until this property is set to a non null value.
* Validation. This allows the Application developer to capture common domain property rules. The rules captured differ for the different property types but are typically Min Value, Max Value for most data types and for strings, Min length, Max Length and a Regular Expression Pattern Match.



Each object will have a unique object identity (ObjectID). In Habanero and Firestarter the nature of this ID is very flexible and the various options are discussed in Object Relational Mapping (Chapter). For our purposes the Object will be modelled with a GUID identifier which is an classical ObjectID. The property modelled for it is by convention called the InvoiceLineID.



For each object you can also identify alternate keys. This is a single or a group of properties that are unique for the Business Object and act as a constraint to ensure that two Business Objects cannot be persisted with the same values for the alternate key. E.g. An invoice cannot have two invoice lines with the same invoice line number.



The last important value to be modelled in Firestarter is the relationships between two business objects. We will show the relationship between Invoice and Invoice line namely.

* An Invoice has one or more Invoice Lines. (Multiple)
* If an Invoice is deleted then all its Invoice lines must be deleted. (Delete Related)

To model these we create a relationship between Invoice and Invoice Line. This relationship represents the rules above. The Order By and Property and Related property will be explained in detail in the chapter on Object Relational Mapping (Chapter xx)

