# 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. The result? A single shared language between the programmer, the program and the business. This separation allows much greater scope for the programmer to implement the customer’s vision.

Do overview of AGILE/waterfall/domain

Pitfall: Over modelling & analysis paralysis

Urban legend tell of developers creating perfectly wallpapered rooms the paper provides from printouts from the domain model with nary a line of code written. Such situation are obviously to be avoidefd but how?.....

Two extremes in programming:

The waterfall approach: requirements analysis, design, (fill in)

The sucking oozying swamp approach: no or very little documented analysis & design

Recommended: Just in time (JIT) analysis & design. Its something which we have all heard of but many fail to stick to? So whats the problem…..

Agile model drive dev

Growing veg versus building a bridge; make compost not girders

AGILE V MDA (Academic)

Impedance mismatch

**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.

### Building the domain

This is when Firestarter comes into its own. 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 without having to focus on the complexities of;

* object-Relational mapping
* security implementation
* business rule implementation.

This frees the developer up to concentrate on the modelling and developing the core business factors which include:

* knowledge of the customer on what he/she wants
* experience of the team in the domain.
* experience of the team with the technologies
* physical, business and emotional relationship with the customer
* experience that the team has working together
* inherent risk of the project (i.e. an aircraft control system is higher risk than a library admin system).

This is the essence of the domain model; its ability to offer something which most software neglects: the opportunity to build a great relationship and engage in genuine collaboration between the client and the developer.

### Existing domains

In the case of existing domains, if you are replacing an existing system and making virtually no changes to the functionality of the system, you can create a complete domain model up front and implement this domain model.

Your model would typically contain at least a class diagram (if you are using a relational database then an Entity Relationship Design). You can implement the database design and reverse engineer the classes from the database design using Firestarter’s powerful reverse engineering capabilities. The Business Objects, relationships, properties and Object to relational mappings will all be modelled for you using the Firestarter wizard.

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 requirements 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

### Detailed domain: Using AGILE methods to step into the unknown

Firestarter has been designed to be used in the most agile manner possible. 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. Having this agility is something which could prove crucial when you are developing a system for an unfamiliar business application or a scenario where the customer is unsure of what they want. These are real life scenarios and occur all the time. So what to do?

In this scenario you should create a detailed model of only the smallest part of the system under development, iterating between Analysing Modeling, Developing and Testing with the customer Stakeholders (user/customer/business analyst) as quickly as possible. In addition, following the agile principle of delaying detailed design until needed, you need only detail the domain model as required. Firestarter has been designed to be used in the most agile manner possible. 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 is to work through an example.

### Use case

First, we create our use case. To the initiated this is a sure fire way to avoid mistakes. In a nutshell, a use case describes the process, whatever that process may be. For example if the client wants a feature where a user clicks an icon and a form is printed this is a process and all the detail surrounds this feature must be recorded in the form of a Use Case.

An example of a typical use case template is featured below. Typically the analyst records the client’s requirements in the use case document. This template serves as an essential communications document for the developer and also serves as the basis for acceptance testing. Ignore this document at your peril! DO A USE CASE

Implement test 1

Model a business object required

Generate

Expand test 1

Add to model

Generate

|  |  |  |
| --- | --- | --- |
| Add order | | |
| Use Case Last Updated: | | 26 March 2008 |
| Actor: | | Standard User |
| Description: | | Actor Adds or Edits order in the system. |
| Assumptions: | | None |
| Pre-Condition: | | None |
| Post Condition: | | None. |
| Trigger: | | * It is time to start work on an order. |
| **Normal Course** | | |
|  | Open the order management section of the system | |
|  | The system will display all orders | |
| **Add a order to the system** | | |
|  | Select **Add** | |
|  | Open a blank order form with the following details:   * Order # - Unique number generated by the system * Order Description * Date Initiated - The date that the matter was accepted by the Order department | |
|  |  | |

Once you create the Use case usenote is complete we open Firestarter.

* Create a new project (Do no have any spaces in project name)
* Create a new class – name it order
* Click on properties – click new – name OrderId – choose GUID( Globally unique identifier)
* Add fields
* Define primary key – select ordered (Add any other unique or alternate keys at this stage
* At this point we do not define any relationships as we working with only one class
* At this point we model the user interface. We leave the UI name as default. Define our UIS (user interfaces)
* Choose Multi select – choose - choose form - choose multi select – we are now done with Firestarter
* Choose generate code – then choose save (Tip do not choose save & generate)
* Choose Habanero path - C:\Program Files\Chillisoft\Habanero v2.1.2\bin

Mapping between code & db

UI definitions for class

A working visual studio solution

……

*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.*

*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.

The following example details typical business objects which form the domain model:

* 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.

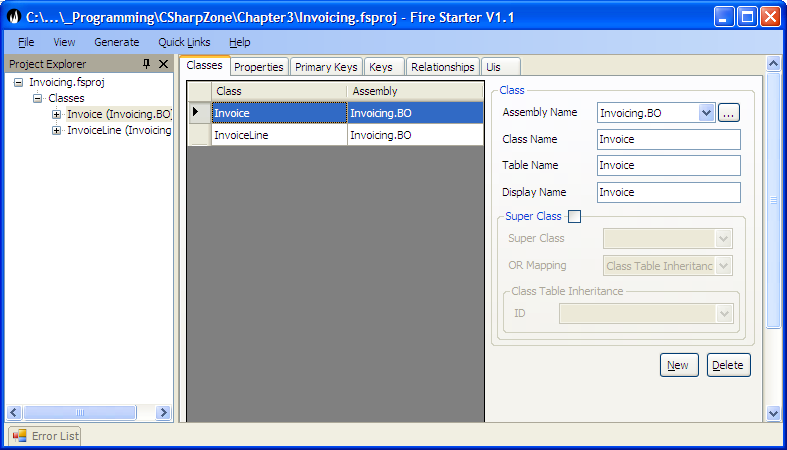
There are many more rules for this system but these have been selected as examples of rules that are modelled in the domain layer. In addition the system has been separated into separate sub systems as per this diagram. For the purposes of this example the Customer and all related rules and data have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



**Capturing the Domain Model using Firestarter**

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.

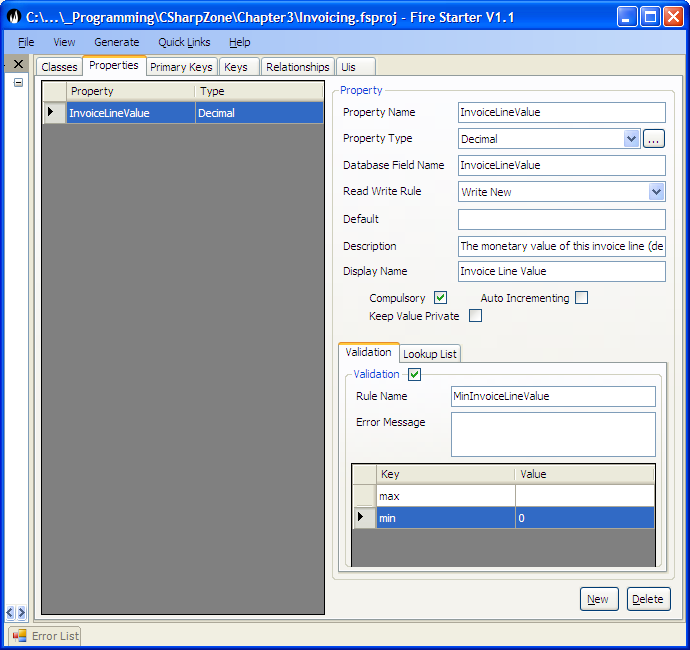


Then we model the various properties that each Business Object has in our case the Invoice Line has a line number – the sequence number of the line printed on the invoice, a line description – a textual description of the invoice line and a value the amount that the customer is being charged.

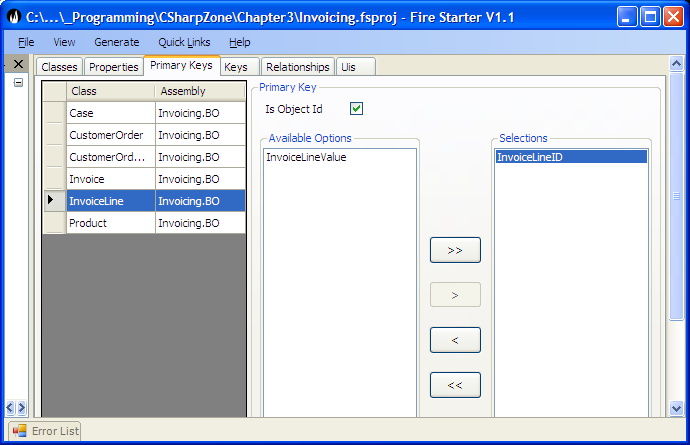
Each of these properties of a business object can have rules set for it. We will demonstrate this by adding the property *Invoice Line Value* to *Invoice Line*.

From this we can see that the *InvoiceLineValue* is

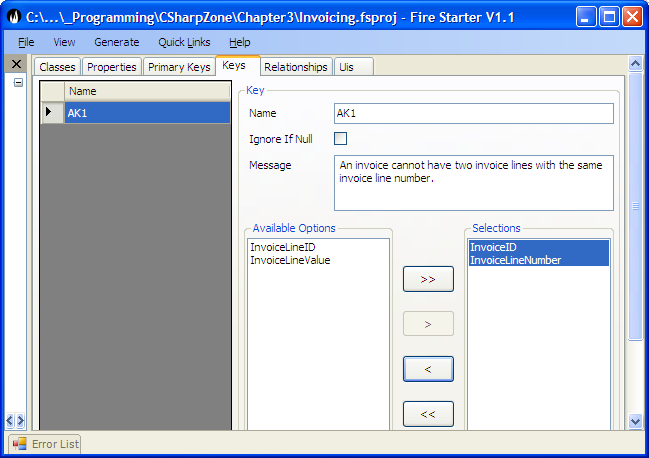
* Type Decimal.
* Read Write Rule of Write New i.e. it must be set to a valid value before the object can be persisted for the first time else the object will be in an invalid state. The *InvoiceLineValue* can never be updated after the object has been persisted to the database. The other options for 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 a classical ObjectID. The property modelled for it is by convention called the InvoiceLineID.



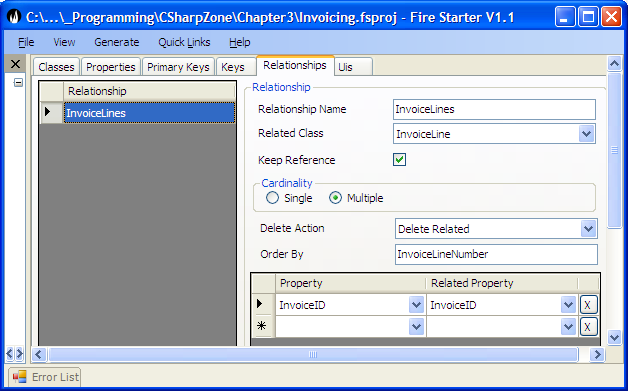
For each object you can also identify alternate keys. This is a single or a group of properties that are unique for the Business Object and act as a constraint to ensure that two Business Objects cannot be persisted with the same values for the alternate key. E.g. An invoice cannot have two invoice lines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



What next? Is this the end