

CHILLISOFT

C SHARP ENTERPRISE ZONE

A GUIDE TO THE HABANERO FRAMEWORK AND CONCEPTS

EARLY ROUGH DRAFT

7/16/2010

CONTENTS

Introduction.....	4
Why this book.....	4
IS THIS BOOK FOR ME?	4
how do i use the book?	4
Book Section Overview	5
Section 1 - Overview:	5
Section 2 – Business Logic Layer:	5
Section 3 - Presentation:	6
Section 4 - Data Access:	6
Section 5 - Miscellaneous:	6
Contributors of the book:	6
Licensing	6
License Details	7
Social Investment	11
CHAPTER 1	12
C Sharp Enterprise Zone	13
Chapter 2: Enterprise Application development concepts.....	13
Development philosophies and practices to value.....	13
Source Control	13
Test driven development.....	14
Continuous Refactoring	14
Continuous integration	15
Agile methodologies and Principles.....	15
Domain Driven Design	15
Infrastructure Challenges	15
Data storage and retrieval	16
Domain modelling and object-relational mapping.....	16
Concurrency control	16

Atomic transactions	16
Application Layering	16
Change	16
References	16
Chapter 3: The Domain Model – Business Objects.....	17
The Business Objects	17
Chapter 4 Domain Model -The Business Object Collection	43
Chapter 5 – Relationships.....	46
Overview.....	46
Using Relationships in Habanero	50
Example	58
Implementation	58
Extending Relationships.....	63
Chapter 6: Creating a Domain Model	63
Building the domain.....	64
Existing domains	65
Detailed domain: Using AGILE methods to step into the unknown	65
Use case	66
Capturing the Domain Model using Firestarter	69

INTRODUCTION

Habanero has been developed over eight years and has adapted to various technologies and development platforms. This architecture was developed by Chillisoft Solutions Pty (Ltd), whose primary revenue stream is, custom application development. Due to the nature of this business, the applications have to be robust, easily extendable, easily deployed, secure, scalable from a single user to thousands of users and easily maintainable by any developer in the company or developers in the customer's company. The framework must also be easily used while retaining extensibility, since our developers' skill range from young graduates to seasoned professionals and our projects range from (2 weeks with a single developer) to 3 years with a large team of developers. Eish! I hear you say... well it was a tall order but one that we have finally mastered enough to share with you.

i.e. Habanero (i.e. a framework that enables development of enterprise applications) is a free and open source allowing developers to review, study, use and contribute to it. In essence Habanero solves the problems that are common to developing and maintaining enterprise applications.

So what constitutes enterprise applications? Enterprise applications, in the broadest sense are applications that create, store and manage information in an organisation. The applications are used in the running of a business where the information is stored in a structured manner over a period of time. The application's information is usually shared between multiple users and is frequently used in geographically distributed locations. Typical examples of enterprise applications are Customer Order Management, Procurement, Asset Management, Financial, Planning scheduling, logistical and supply chain management.

WHY THIS BOOK

As the Habanero development community has continued to grow, it has become clear that there is a need to provide a more extensive guide on the use of Habanero. We have therefore developed a comprehensive book to provide developers with additional information on not only on how to use Habanero and Firestarter but also on the common problems and solutions that are incurred in developing enterprise applications.

IS THIS BOOK FOR ME?

The purpose of the book is to cover all topics that are relevant to enterprise application development in sufficient detail so as to provide the reader with adequate knowledge to develop a robust scalable enterprise application. The book leverages off Chillisoft's development experience allowing the developer to dodge common mistakes. We acknowledge that it is impossible for one book to cover all topics (in sufficient detail), and have consequently provided additional references to other books and articles for those readers who are interested in or who require more detail.

This book is intended for the use of junior and senior developers. The book assumes that you have a reasonable knowledge of object oriented programming and practices, C#, .NET and database management systems. The book also assumes that you have at least some basic knowledge of enterprise application development.

HOW DO I USE THE BOOK?

To help you determine how to read this book we have categorised our audience into two extremes. Most likely you will fit somewhere in between these categories:

- 1) A developer who wants to know how to use Habanero framework to develop an application.

- 2) A developer who wants to understand why and how the framework was developed, how to extend it and or how to use the concepts to develop their enterprise applications.

We understand that the readers of this book and the users of Habanero are busy. We have therefore endeavored in every way to ensure that the book can be used to provide immediate benefit at each stage. Since software developers learn best by examples, we have included extensive examples and a worked example that will be developed through the various sections of the book.

BOOK SECTION OVERVIEW

SECTION 1 - OVERVIEW:

This section provides the generation of a sample application, using Habanero. It introduces all the principles that are required to develop enterprise applications. We would suggest that all readers read these chapters.

Chapter 1 covers the use of Habanero and Firestarter (an application code generator specialised for Habanero). This is very brief and used merely to give the reader a quick overview of the Framework.

Chapter 2 covers the principles and practices of enterprise application development. This chapter will touch on all the concepts that will be covered in the remaining chapters of the book.

SECTION 2 – BUSINESS LOGIC LAYER:

This section covers the development of the business logic layer of an enterprise application, also called the domain model. The domain model involves a group of objects that models the area of business area that your system is implementing. The domain model in the simplest case closely resembles the relational database model but in more complex business scenarios can be significantly different. The domain model provides the layer where you model the domain behavior, data, validation and object relationships. One of the important functions of this layer is to hide the database, so this section will not discuss any database and object persistence issues. Object persistence will be addressed in Section 4.

Each chapter in the section starts with a brief introduction to the topic. This is followed by an example of how to model this using Firestarter and Habanero and how to use this in an application. A developer who merely wants to use Habanero can stop at this point and move onto the next chapter.

The chapter will then delve into the more advanced features of Habanero and Enterprise applications including how Habanero implements these features, the design decisions behind the implementation and where relevant a discussion of why these decisions were made when compared to alternates.

Finally the chapter moves onto how these features of Habanero can be extended to deal exceptional situations and requirements.

SECTION 3 - PRESENTATION:

This section covers the development of the presentation layer. The presentation layer covers user interfaces for capturing, searching, manipulating, reporting on and viewing objects. The technologies covered here include, ASP, C# and Visual WebGui (an AJAX framework for web development). Our preferred reporting tool is Active reports.

SECTION 4 - DATA ACCESS:

This section covers the development of the Data layer i.e. the mapping of business objects to a data store. This section includes persisting objects to XML, memory caches, object data stores and relational databases. Since most enterprise applications persist their data to relational databases, a large part of this section revolves around persisting objects in a relational database (using object relational mapping). The structure of these chapters is the same as section 2 and should be used in the same manner.

SECTION 5 - MISCELANEOUS:

In addition to the presentation of domain objects in a user interface, our business objects may also need to be available to other applications e.g. via web services. In this section we will discuss various solutions for these.

Contributors of the book:

This book has been written and edited by

Peter Wiles (Google Dev)

Brett Powell (Glory Dev)

Soriya Das (Sarky Dev)

Mark Whitfeld (Complex Dev)

Eric Savage (Simply Dev)

Hagashen Naidoo (Stirrer Dev)

Sherwin Moodley (Pretty Dev)


Duncan Hodgson (BreakIT Dev)

Deerasha Singh (Guji Dev)

Dominic Hardman (Money Dev)

LICENSING

Copyright © 2008 Chillisoft Solution Services Pty (Ltd).

C Sharp Enterprise Zone is an intermediate-to-advanced book created using an open-source development process. C Sharp Enterprise Zone is published under the  ([Creative Commons Attribution-NonCommercial-Share Alike 3.0 license](https://creativecommons.org/licenses/by-nc-sa/3.0/))

LICENSE DETAILS

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

"Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

"Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(g) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

"Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

"License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Noncommercial, ShareAlike.

"Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

"Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

"Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

"You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

"Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

"Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights described in Section 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(d), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(d), as requested.

You may Distribute or Publicly Perform an Adaptation only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-Noncommercial-Share Alike 3.0 US) ("Applicable License"). You must include a copy of, or the URI, for Applicable License with every copy of each Adaptation You Distribute or Publicly Perform. You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License. You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(d) may be implemented in any reasonable manner;

provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

For the avoidance of doubt:

Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

Voluntary License Schemes. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

This book is provided free for use to the Habanero community under the Creative Commons License Attribution-Noncommercial-Share Alike 3.0 unported (i.e. please refer to <http://creativecommons.org/licenses/by-nc-sa/3.0/> for additional legal details).

In summary the license caters for:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

SOCIAL INVESTMENT

If you find this book useful and wish to make a financial contribution then please contribute to [A charity for the ???](#)

STILL TO BE ADDED...

CHAPTER 2: ENTERPRISE APPLICATION DEVELOPMENT CONCEPTS

An Enterprise Application in its broadest sense is an application that manages information in an enterprise. Typically they are the systems used in the daily operations of a business and are required to be robust, secure and scalable because a loss of uptime in the system results in a decrease in productivity of its users.

This chapter will first cover the development philosophies and practices that we value in developing enterprise applications, and then outline a set of infrastructure challenges that just about every enterprise application developer encounters. These challenges will be expanded on in the rest of the book.

DEVELOPMENT PHILOSOPHIES AND PRACTICES TO VALUE

Over the years of delivering custom developed enterprise applications we have come to value certain development methodologies and philosophies very highly. These are important to cover as they have deeply affected the way Habanero was put together, and if you are to use Habanero in any capacity they are philosophies we hope you will value too.

SOURCE CONTROL

For every kind of development using some form of source control is an absolute must. Even if you're developing software as a one-person team a source repository is essential as it brings with it many advantages, such as:

1. It allows you to undo program-breaking changes easily. Because of this it allows you to experiment with an idea knowing that you can always revert if things get out of hand
2. It records changes made to the source code over time, allowing you to retrieve the source of a project as at a particular date, or to see exactly when a particular change was made (and by whom if necessary)
3. It allows multiple people to work on the same set of code at the same time without overwriting each others' changes
4. It allows you to release a bug fix for the current live version of your software even while you're developing new functionality
5. It becomes the de-facto master-copy of your code, which along a solid backup process means you always know that your source code is safe and sound (not on someone's computer somewhere...)

Of course there are many ways that source control can be used and an astounding selection of tools that can be utilized. The important thing is that source control is used, not the particular tool to accomplish it.

At Chillisoft we use Subversion for all our source control needs and we've found it to be a top class solution as well as being free and open source which means it has the advantage of being supported by many other open source tools. To serve the repository we use VisualSVN Server and as our standard front end we use TortoiseSVN (no Visual Studio integration though, we have never found that to be a particularly helpful thing).

Some of the big distributed open source projects like Mozilla and Linux Kernel are using distributed version control systems. If your team is geographically distributed it might make sense to go with something that supports this kind of structure better, like Git.

FURTHER READING

For the what, why and how of Subversion read *Pragmatic Version Control Using Subversion (2nd Edition)* by Mike Mason [Mason].

For the distributed world the guys at the Pragmatic Bookshelf have put together a similar book for Git called *Pragmatic Version Control Using Git* by Travis Swicegood [Swicegood].

TEST DRIVEN DEVELOPMENT

If there is a single practice that has dramatically changed the face of software development over the past decade it is Test Driven Development. It is a simple-to-explain concept but it leads to a paradigm shift in the way one develops and thinks that we believe will become the baseline standard for development in the years to come.

It is often noted that the cycle of testing, debugging and correcting takes a large majority of a traditional project's development time. This is largely because a lot of testing is done manually by a team of testers. These testers in turn provide feedback to the developers, who correct the flaws and pass the system back to the testers. Once again the testers must test the system in its entirety to ensure that no new bugs have been introduced. And so, slowly, the bug count drops to the goal of zero.

It's immediately obvious that some sort of automated testing process would substantially speed up this cycle. The question has always been how to do this effectively without extending project *development time* writing code that doesn't add functionality. Tools, such as the industry standard xUnit (JUnit in Java, NUnit in .NET) have helped with this because they remove most of the overhead involved in setting up test suites. But more importantly the practice of test-first development, or test-driven development, affects the way systems are designed (for the better) and results in a significant decrease in the length of that testing, debugging and correcting cycle.

TODO: basic TDD structure, Red Green Refactor.

Habanero and all production projects at Chillisoft are developed using a test-driven development philosophy. At the time of writing this there are over 6000 automated tests for Habanero itself, which means that we can be confident that in the process of creating new functionality we are not breaking other features.

FURTHER READING

The original, seminal book on TDD is *Test-Driven Development: By Example*, by Kent Beck [Beck]. It is a basic introduction to the concepts and contains a really good extended example of developing using TDD. We recommend being extremely rigorous with the principles laid down by Beck in this book.

Once you've mastered the basics, Gerard Meszaros' *xUnit Test Patterns: Refactoring Test Code* is an excellent book on real-world testing philosophies and patterns.

CONTINUOUS REFACTORING

Refactoring is another practice that has entered the developer's vernacular in the past decade and become standard. Martin Fowler coined the term and defined a refactoring as "a change made to the internal

structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [Fowler, 53]. Refactoring is changing code structure, but it is not altering behaviour.

A constant vigilance for the smells of bad code is required if we are to stay ahead of code entropy. Because of this the steps of test-driven development include a refactoring step after each passing test. Its one thing to say “this code needs refactoring, I’ll add that to the to-do list,” and another to continuously refactor things little by little so that those big refactoring jobs never make it onto the to-do list. We have realized that with each line of code written our programs move closer to chaos unless this practice of continuous refactoring is applied. Areas of our programs become no-go zones that no one can maintain or improve upon unless we’re constantly applying the rule of not allowing software entropy to gain a foothold.

Refactoring without automated tests is a dangerous activity so the practices of test-driven development and continuous refactoring go hand-in-glove; one cannot be practiced without the other.

Many refactorings can be performed better by automated tools (or at least the substeps can), and we highly recommend using a plugin for Visual Studio like Resharper to make continuous refactoring that much easier.

FURTHER READING

The original book on Refactoring is Martin Fowler’s *Refactoring: Improving the Design of Existing Code*. In it Fowler has made a catalog of standard refactorings that outline step by step how to perform things like extracting a method or moving a method without breaking other code. He also covers a variety of “code smells” and possible refactorings to apply in these cases to alleviate the smell.

Refactoring to Patterns by Joshua Kerievsky builds upon Fowler’s book by showing how you can refactor existing code into well structured patterns. Test-driven development advocates a practice of less design up front (countering the ills of applying patterns before they’re need), while refactoring to patterns shows us how we can still end up with a pattern based design via refactoring.

CONTINUOUS INTEGRATION

AGILE METHODOLOGIES AND PRINCIPLES

DOMAIN DRIVEN DESIGN

INFRASTRUCTURE CHALLENGES

There are a set of challenges that just about every enterprise application is required to overcome. In this section we will go over these one by one, discussing what the challenge is, when and why it needs to be overcome and briefly how this can be achieved by using Habanero. The infrastructure challenges discussed here are particular to the design philosophies outlined above, but many will apply given a different set of design philosophies too.

DATA STORAGE AND RETRIEVAL

DOMAIN MODELLING AND OBJECT-RELATIONAL MAPPING

CONCURRENCY CONTROL

ATOMIC TRANSACTIONS

APPLICATION LAYERING

CHANGE

REFERENCES

[Mason]

Mason, Mike. *Pragmatic Version Control Using Subversion (2nd Edition)*. Raleigh, North Carolina: The Pragmatic Programmers LLC, 2006. ISBN: 0-9776166-5-7 (<http://pragprog.com/titles/svn2>)

[Swicegood]

Swicegood, Travis. *Pragmatic Version Control Using Git*. Raleigh, North Carolina: The Pragmatic Programmers LLC, 2008. ISBN: 9781934356159 (<http://pragprog.com/titles/tsgit>)

[Beck]

Beck, Kent. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.

[Meszaros]

Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code*. Boston MA: Addison-Wesley

[Fowler]

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 2000. CBS 87.

[Kerievsky]

CHAPTER 3: THE DOMAIN MODEL – BUSINESS OBJECTS

MAKING A START

“There is no abstract art. You must always start with something. Afterward you can remove all traces of reality.”.... Pablo Picasso (Spanish Cubist painter (1881 - 1973))

The heart of a well designed enterprise application is a good Business Domain model. This is why we start here.

The mapping of the rich domain model to the data store, the implementation of complex use cases using the process logic layer and the development of the user interface layer will be discussed later in relation to the domain model. Habanero provides the framework for developing, maintaining and refactoring a rich domain model. Firestarter provides a tool for rapidly implementing and managing the domain model in code.

A good understanding of business objects is a pre-requisite to understanding domain models and domain driven design (which essentially is the modeling of business or domain objects and their relationships).

THE BUSINESS OBJECTS

In a well designed object oriented system, there are many objects that collaborate to implement any functionality. So what differentiates a business object from other objects in the system?

A business object:

- Is an abstract representation of an entity or concept that exists in the business problem domain, e.g. An invoicing system will contain concepts such as Invoice, Invoice Line
- Has a state that exists beyond the life cycle of a single instance of the application i.e. a business object can be persisted or retrieved from a data store.
- Implements business rules related to the validation, editing and persisting of the business objects data.
- Doesn't know how it will be used i.e. it can be used by a Windows Form, ASP or a web service.
- Encapsulates its data and implements business rules so that it protects the validity of its state independently of the client using it, i.e. if the business rule is that the Invoicing Date must be less than today, the Business Object will enforce this rule.
- Encapsulates any security rules relating to it, e.g. only a user with profile 'Invoicing User' can create a new invoice.
- Has behavior i.e. the business object implements functionality expected of the entity in the domain e.g. an invoice might have a method to Calculate Days Overdue for payment, Amount due for payment.

- Has an identity. The identity of a business object is an important issue because it provides the ability to persist and retrieve the business object from a data store. Ideally the Identity should be globally unique and immutable.

EXAMPLE OF A BUSINESS OBJECT

Let's take a look at how this is implemented in Habanero. We will start with a business object called Customer, and for now the only property that customer will have is customer name. Customer name is compulsory and must have a length greater than 5 characters and less than 100. For the purposes of explaining the use and functionality of the Customer Class we will be using Unit tests developed using the NUnit Framework. The Full Code for the Customer class and the Customer Test class used are available from the book downloads Chapter 3\Example1\ Customer.sln.

The FireStarter Project set up in Chapter 3\Example1\Customer.fsproj.

All the Customer object code is generated using Firestarter and is thus generated into the Customer.Def Partial Class.

In the Customer.Def.cs you will see the following implementation. The customer inherits from the Business Object Class (following the Layer Supertype Pattern – Fowler - xxx).

```
public partial class Customer : BusinessObject
{
    #region Properties
    public virtual String CustomerName
    {
        get
        {
            return ((String) (base.GetPropertyValue("CustomerName")));
        }
        set
        {
            base.SetPropertyValue("CustomerName", value);
        }
    }

    public virtual Guid? CustomerID
    {
        get
        {
            return ((Guid?) (base.GetPropertyValue("CustomerID")));
        }
        set
        {
            base.SetPropertyValue("CustomerID", value);
        }
    }
    #endregion
}
```

As you can see this class contains two properties: the CustomerName and the object's ID property CustomerID.

In addition to the Customer Class we have a ClassDefs.xml file. Although the application developer does not need to understand the details of this file, it is important to understand that it exists and what it does. The

ClassDefs.xml assists Habanero in implementing the Meta-Data Mapping Pattern (See Fowler p. 306) and Meta Programming paradigm.

In the ClassDefs.xml file you can see the class Customer is defined with its two properties CustomerName and

```
<classes>

  <class name="Customer" assembly="Customer.BO">

    <property name="CustomerName" />

    <property name="CustomerID" type="Guid" readWriteRule="WriteNew"
displayName="CustomerID" />

    <primaryKey>

      <prop name="CustomerID" />

    </primaryKey>

  </class>

</classes>
```

CustomerID. CustomerID is defined as the PrimaryKey (Object ID).

USING THE CUSTOMER BUSINESS OBJECT

From here we are going to show the basics of how to use a Customer Business Object, and the functionality and capability that it provides to the application developer. Most of this functionality will be demonstrated using the appropriate unit tests in [TestCreateCustomer](#).

CREATING A NEW CUSTOMER

```
[Test]
public void TestCreateNewCustomer_NoPropertyRules()
{
    //-----Set up test pack-----

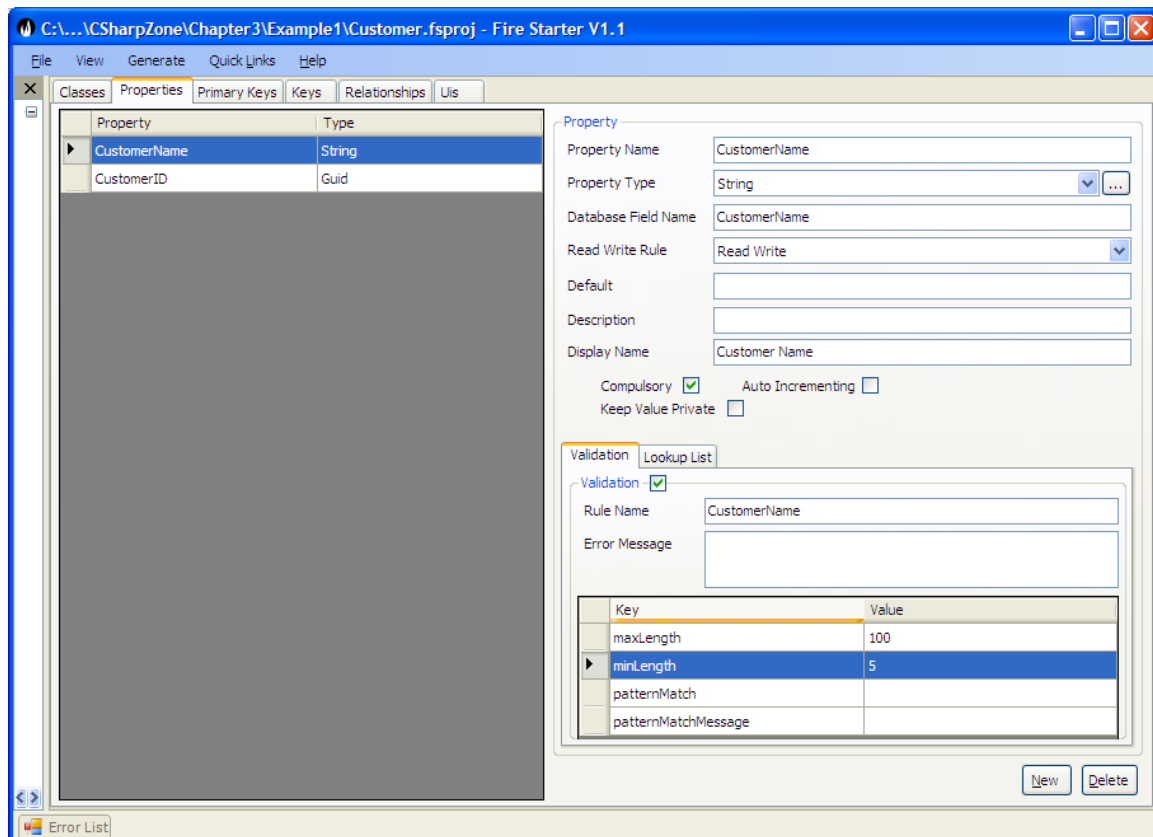
    //-----Assert Precondition-----

    //-----Execute Test -----
    Customer customer = new Customer();
    //-----Test Result -----
    Assert.IsTrue(customer.Status.IsNew);
    Assert.IsNotNull(customer.CustomerID);
    Assert.IsFalse(customer.Status.IsDeleted);
    Assert.IsFalse(customer.Status.IsDirty);
    Assert.IsFalse(customer.Status.IsEditing);
    Assert.IsTrue(customer.Status.IsValid());
}
```

From the above you can see that a new Business Object Customer is created with a status IsNew. Habanero has also already created an object ID for the Customer. The business object has not yet been edited and it's IsDirty and IsEditing Status is therefore false. No rules have yet been set up for the Customer Business Object – its IsValid property is therefore true.

CREATING A NEW CUSTOMER WITH CUSTOMERNAME COMPULSORY

The Business rule that the customer name is compulsory is set in FireStarter along with the minimum (5 characters) and maximum length (100 characters) of the CustomerName.



This does not in any way change the Customer.Def.cs but instead changes the metadata rules set up in the ClassDef.xml.

The Customer name property definition is modified to be:

```
<property name="CustomerName" compulsory="true">
  <rule name="CustomerName">
    <add key="maxLength" value="100" />
    <add key="minLength" value="5" />
  </rule>
</property>
```

The Habanero Framework implements the Customer Compulsory rule as demonstrated in the following tests.

```
[Test]
public void TestCreateNewCustomer_CustomerNameCompusory()
{
    //-----
    //When a new customer is created it is created with
    // broken rules for any
    // compulsory properties that do not have a default value set
    // (In this case Customer name).
    //-----
    //-----Set up test pack-----

    //-----Assert Precondition-----

    //-----Execute Test -----
    Customer customer = new Customer();
    //-----Test Result -----
    Assert.IsTrue(customer.Status.IsNew);
    Assert.IsNotNull(customer.CustomerID);
    Assert.IsFalse(customer.Status.IsDeleted);
    Assert.IsFalse(customer.Status.IsDirty);
    Assert.IsFalse(customer.Status.IsEditing);
    Assert.IsFalse(customer.Status.IsValid());
    StringAssert.Contains("'Customer Name' is a compulsory field and has
no value",
        customer.Status.IsValidMessage);
}
```

SET PROPERTY VALUE

The Habanero Framework implements the CustomerNameRule as demonstrated in the following tests.

```
[Test]
public void TestNewCustomer_SetCustomerName_ToValidValue()
{
    //When a property is set to a valid value for a compulsory field
    // that has a broken rule the rule is set to no longer broken.
    //-----Set up test pack-----
    Customer customer = new Customer();
    //-----Assert Precondition-----
    Assert.IsTrue(customer.Status.IsNew);
    Assert.IsFalse(customer.Status.IsDirty);
    Assert.IsFalse(customer.Status.IsEditing);
    StringAssert.Contains("'Customer Name'
is a compulsory field and has no value",
        customer.Status.IsValidMessage);

    //-----Execute Test -----
    customer.CustomerName = "Valid Name";

    //-----Test Result -----
    Assert.IsTrue(customer.Status.IsDirty);
    Assert.IsTrue(customer.Status.IsEditing);
    Assert.AreEqual("", customer.Status.IsValidMessage);
}
```

THE TEST CLASS CONTAINS SOME ADDITIONAL TESTS THAT DEMONSTRATE OTHER VARIANTS OF SETTING AND GETTING PROPERTY VALUES, AND EXAMPLES OF HOW SETTING PROPERTY VALUES CHANGES THE CUSTOMER STATUS BASED ON THE PROPERTY RULES.

SAVING BUSINESS OBJECTS

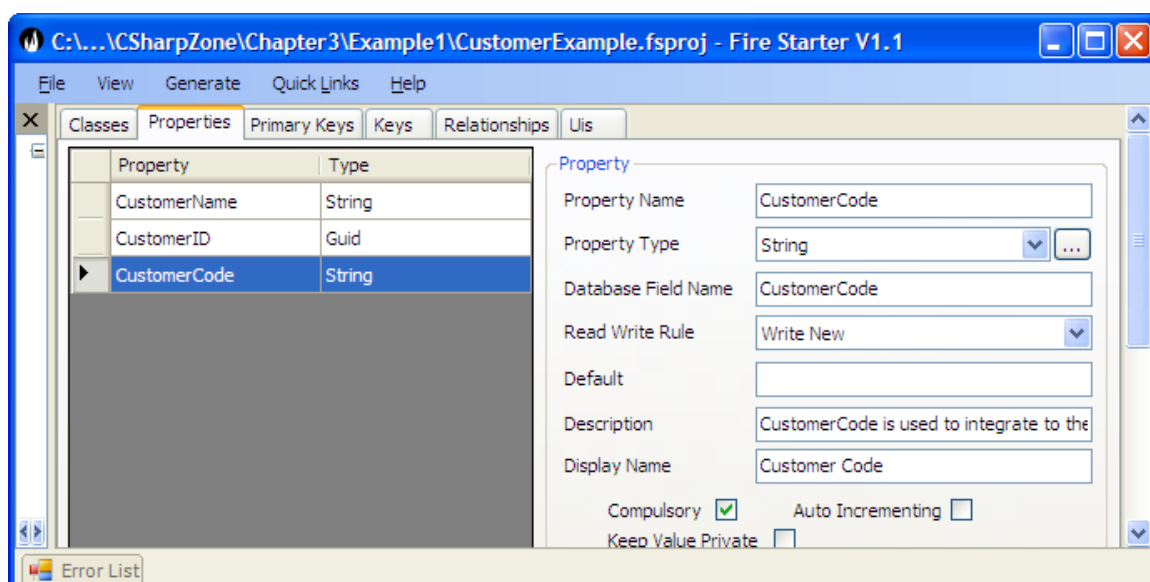
The implementation details of saving and loading Business objects from a DataStore are discussed in detail in section 4. For the purposes of this chapter we are just going to look at some methods to save a single object to a DataStore. The tests used in these examples are all from the `TestSaveCustomer`. It is clearly evident from this test that a saved business object is no longer dirty and is no longer new. The test `Test_Save_Invalid_ValidCustomer` illustrates that a business object in an invalid state cannot be persisted and the error `BusObjectInAnInvalidStateException` will be thrown if the application tries.

```
[Test]
public void Test_SaveValidCustomer()
{
    //-----Set up test pack-----
    Customer customer = new Customer();
    customer.CustomerName = "Valid Name";
    customer.CustomerCode = "Code";
    //-----Assert Precondition-----
    Assert.IsTrue(customer.Status.IsNew);
    Assert.IsTrue(customer.Status.IsDirty);
    //-----Execute Test -----
    customer.Save();
    //-----Test Result -----
    Assert.IsFalse(customer.Status.IsNew);
    Assert.IsFalse(customer.Status.IsDirty);
}
```

`Test_Save_Invalid_ValidCustomer` - do test in code

USING PROPERTY READ WRITE RULES

One of the other rules that is often implemented for a Business Object Property relates to the reading and



writing of a property depending on the status of the business object. For instance, a customer has a property `CustomerCode` whose value is used to integrate to the External systems. This property is compulsory and can never be changed once it has been set. We add this property to `FireStarter` with the ReadWrite Rule – `WriteNew`.

The other available read-write options are:

- **Read Only** – used when a properties value is loaded from a data store but is never updated to the business object e.g. if a stored procedure or external application updates this property's value and it is only ever read from the datastore by the business object and can never be set, updated or persisted back.
- **Read Write** – The property's value can be read and updated many times in the life of the Business Object. This is the default setting.
- **Write New** – The property's value can only be set when the object is new (i.e. the business object has never previously been updated to a data store).
- **Write Not New** – A value can never be set if the business object is new. The value can be set at any time that the business object is not new.
- **Write Once** – A value for a property can only be set to a business object once. This may be set at any stage of the business object's life but once the value for the property is set it can never be changed.

There are a few tests in the `TestSetCustomerBOProp` class relating `ReadWriteRules` but the most pertinent test is illustrated below. The test shows that the `BusinessObjectReadWriteRuleException` error is raised if the application attempts to update the Business Object property in contravention of its `ReadWrite Rules`. The details of how this is implemented are shown in `Implementing Business Objects` below

```
[Test]
public void Test_SetCustomerCode_ForPersistedCustomer()
{
    //A WriteNew property can not be written to when the business object
    //already persisted.
    //-----Set up test pack-----
    Customer customer = CreateSavedCustomer();
    IBOProp customerCodeBoProp = customer.Props["CustomerCode"];
    //-----Assert Precondition-----
    Assert.AreEqual(PropReadWriteRule.WriteNew,
        customerCodeBoProp.PropDef.ReadWriteRule);
    Assert.IsFalse(customer.Status.IsNew);
    //-----Execute Test -----
    try
    {
        customer.CustomerCode = "Code New";
        Assert.Fail("expected Err");
    }
    //-----Test Result -----
    catch (BusinessObjectReadWriteRuleException ex)
    {
        StringAssert.Contains("Error writing to property 'Customer Code'
            because it is configured as a 'WriteNew' property", ex.Message);
    }
}
```

Customer Lookup Rule

TODO: Needs section of using a lookup list for a property.

DELETING A BUSINESS OBJECT

Deleting a Business Object is extremely simple. First the object is marked for deletion using `customer.Delete()`; and then the deletion is persisted to the DataStore using `customer.Save()`; The only thing to remember is that when a customer is set as Deleted this has not yet been persisted to the DataStore – only when the `customer.Save()` is called is the object actually deleted. This is implemented so as to allow objects marked for deletion to be added to a transaction along with other objects. (See Section 4: Data Access Layer -Transaction Committer).

```
[Test]
public void Test_DeleteCustomer()
{
    //-----Set up test pack-----
    Customer customer = CreateSavedCustomer();
    customer.Delete();

    //-----Assert Precondition-----
    Assert.IsFalse(customer.Status.IsNew);
    Assert.IsTrue(customer.Status.IsDeleted);
    Assert.IsTrue(customer.Status.IsDirty);

    //-----Execute Test -----
    customer.Save();

    //-----Test Result -----
    Assert.IsTrue(customer.Status.IsNew);
    Assert.IsTrue(customer.Status.IsDeleted);
    Assert.IsFalse(customer.Status.IsDirty);
}
```

BUSINESS OBJECT IDENTITY

Business object identity primarily impacts the Data Access layer. However it should be noted that it has important consequences in the Business Object Layer, hence the reason why it is briefly discussed here (i.e. a more detailed discussion can be found in Section 4 – Data Access Layer). The managing and tracking of the Identity of an object is essential for any persistable object (i.e. any object that persists its state beyond the life of a single instance of an application), because the object identity is used to retrieve the object from the external datastore. It is thus important to implement a strategy for managing object identity since all business objects are persistable.

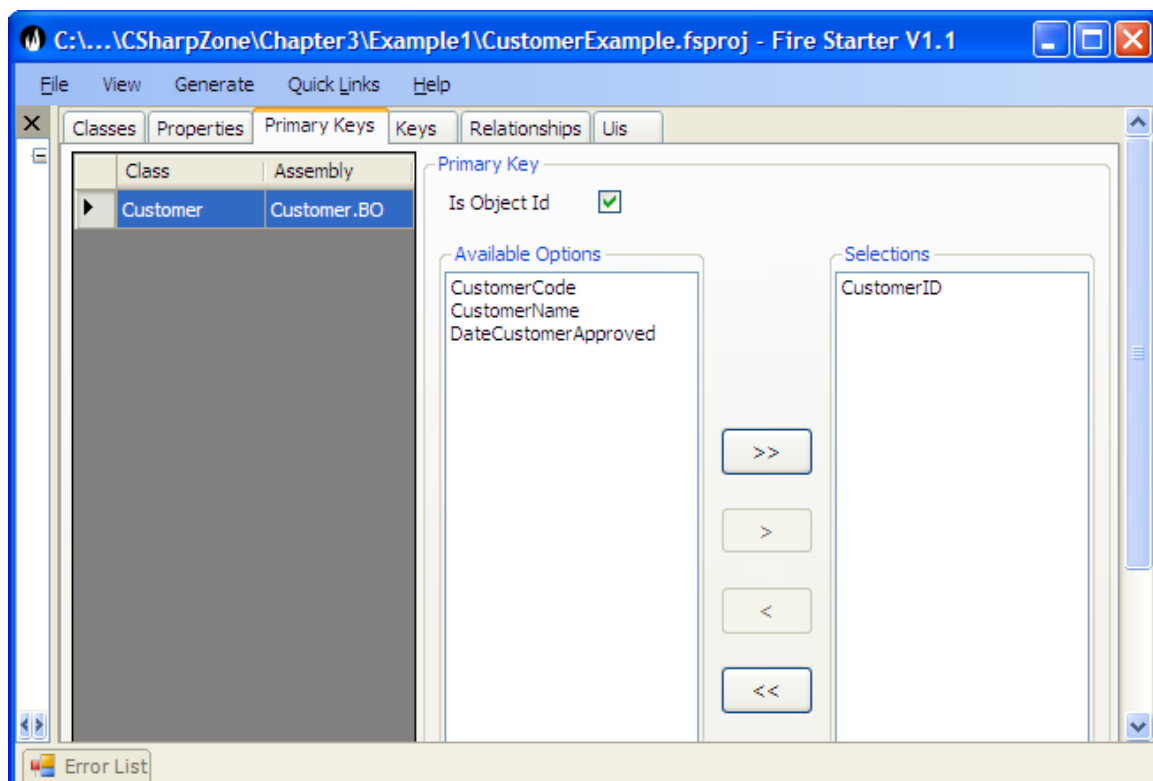
Principles of Object Identity:

1. Ideally an object identity should be immutable. An object identity that can be modified creates a set of problems for managing an application, particularly concurrency issues. For example, User1 and User2 load the same object from the datastore. User1 updates the object's identity and persists it to the database. Then User2 edits some other properties on the object and updates it. The original object cannot be found in the data store because its identity has been changed.

2. The object identity should be a globally unique identifier. This ensures that business objects can be found and distributed while only being identified by their identities, independently of the application, class or type of object.
3. Object identity should have no meaning. The object should be identified and stored, based on a value that has no meaning in the business domain. Properties such as ProductCode, should not serve as the identifier of an object even if they are immutable. The reason is simply that at some point the 'immutable' value is going to be changed. Typically, a user is going to type in the incorrect code and this is going to have to be edited or the organisation is going to modify the algorithm for generating ProductCode and when this happens the ProductCode will be changed.
4. From the above we can conclude that the Business Object's identity should not be a composite of more than one value e.g. Surname, Firstname should not be used to provide a composite identity.

From the above, we can conclude that in the ideal case where we have control or influence over the database design we can implement these principles. Unfortunately we have found that there are many cases when these principles cannot be adhered to. A typical example is when rewriting an existing system, where the system already has a database that cannot be changed. Consequently, the Habanero Framework provides strategies whereby you can circumvent any of these object identity principles when developing an application. It is however important to point out that if you choose to use a mutable property for the object identity you should implement an appropriate concurrency control strategy for dealing with this. See Section 4 – Data Access Layer – Concurrency Control.

The Object Identity is modelled in FireStarter. The Object identity is always composed of one or more properties of the Business Object. Ideally you will be able to create a property (and field in the database) specifically for this purpose – in this example CustomerID property.



LOADING A BUSINESS OBJECT FROM THE DATASTORE

In cases where the application knows the ID, the business object is easily retrieved from the data store. All object retrieval is carried out using the `BusinessObjectLoader`

(`BORegistry.DataAccessor.BusinessObjectLoader`). The details that are involved in Loading

```
[Test]
public void Test_LoadCustomerUsingID()
{
    ///This test shows that if a persisted object is loaded from the
    /// dataStore using the BusinessObjectLoader.GetBusinessObject.
    /// Then an object with the exact same status and data as
    /// the persisted object is loaded.
    ///-----Set up test pack-----
    Customer customer = CreateSavedCustomer();

    ///-----Assert Precondition-----
    Assert.IsFalse(customer.Status.IsDirty);
    Assert.IsFalse(customer.Status.IsNew);
    Assert.IsTrue(customer.Status.IsValid());

    ///-----Execute Test -----
    Customer customer2 =
        GetBusinessObjectLoader().GetBusinessObject<Customer>(customer.
        ID);

    ///-----Test Result -----
    Assert.IsFalse(customer2.Status.IsDirty);
    Assert.IsFalse(customer2.Status.IsNew);
    Assert.IsTrue(customer2.Status.IsValid());

    Assert.AreEqual(customer2.CustomerCode, customer.CustomerCode);
    Assert.AreEqual(customer2.CustomerName, customer.CustomerName);
    Assert.AreEqual(customer2.CustomerID, customer.CustomerID);
    Assert.AreSame(customer, customer2);
}
```

Business Objects from a database are discussed in Detail in Section 4 – Data Access Layer.

It is important to note that in cases such as this where a reference to the original customer is still being held by the application the second customer returned by the `BusinessObjectLoader` (i.e. `customer2`) is in fact the exact same object as the original customer i.e. `Assert.AreSame(customer, customer2)`. The implementation of this will be discussed in Implementing Business Objects Loading below.

You can also retrieve a business object from the `BusinessObjectLoader` using a criteria (e.g. `CustomerCode = 'HO13787'`). This is frequently used when loading a collection of business objects but can also be used when loading a particular business object. Once again, the test shows that *exactly* the same instance of the customer

```
[Test]
public void Test_LoadCustomerUsingStringCriteria()
{
    ///This test shows that if a persisted object is loaded from the
    /// dataStore using the BusinessObjectLoader.GetBusinessObject.
    /// Then an object with the exact same status and data as
    /// the persisted object is loaded.
    ///-----Set up test pack-----
    Customer customer = CreateSavedCustomer();
    ///-----Assert Precondition-----
    Assert.IsFalse(customer.Status.IsNew);
    ///-----Execute Test -----
    string loadCriteria = "CustomerCode = " + customer.CustomerCode;
    Customer customer2 =
        GetBusinessObjectLoader().GetBusinessObject<Customer>(loadCriteri
        a);
    ///-----Test Result -----
    Assert.IsFalse(customer2.Status.IsNew);
    Assert.AreSame(customer, customer2);
}
```

is returned by the BusinessObjectLoader.

The Criteria objects and criteria strings are significantly more powerful than shown so far and includes all the normal operators that are required i.e. AND, OR, opening and closing brackets, <>, =, <=, >=, <, >, IS NOT, IS, NOT LIKE, LIKE.

Note:

- If you try loading an individual object (GetBusinessObject) with a criteria and more than one matching object is found in the database an error will be raised.
- The CustomerCode is the Business Object's property name and not the database field name. This is very important because by working only with the domain model the application developer is totally insulated from the database. The Habanero Framework interprets the property name into the appropriate field name when querying the datastore.

Once the object is loaded it can be refreshed multiple times from the database. An object that is currently being edited (customer2.Status.IsDirty == true) may not be refreshed but a business object that is not dirty can be refreshed. This is very useful when you are about to carry out a calculation and want to ensure that you have the latest version of the object (i.e. Another user may have edited the object since you loaded it). For more details on concurrency control see Section 4 – Data Access Layer – Concurrency Control.

The object is refreshed using the GetBusinessObjectLoader().Refresh(customer) method.

CANCELLING EDITS ON AN OBJECT

If the application or user has made an error then edits (changes to the object not yet persisted to the datastore) can be cancelled using the Restore Method on the business object.

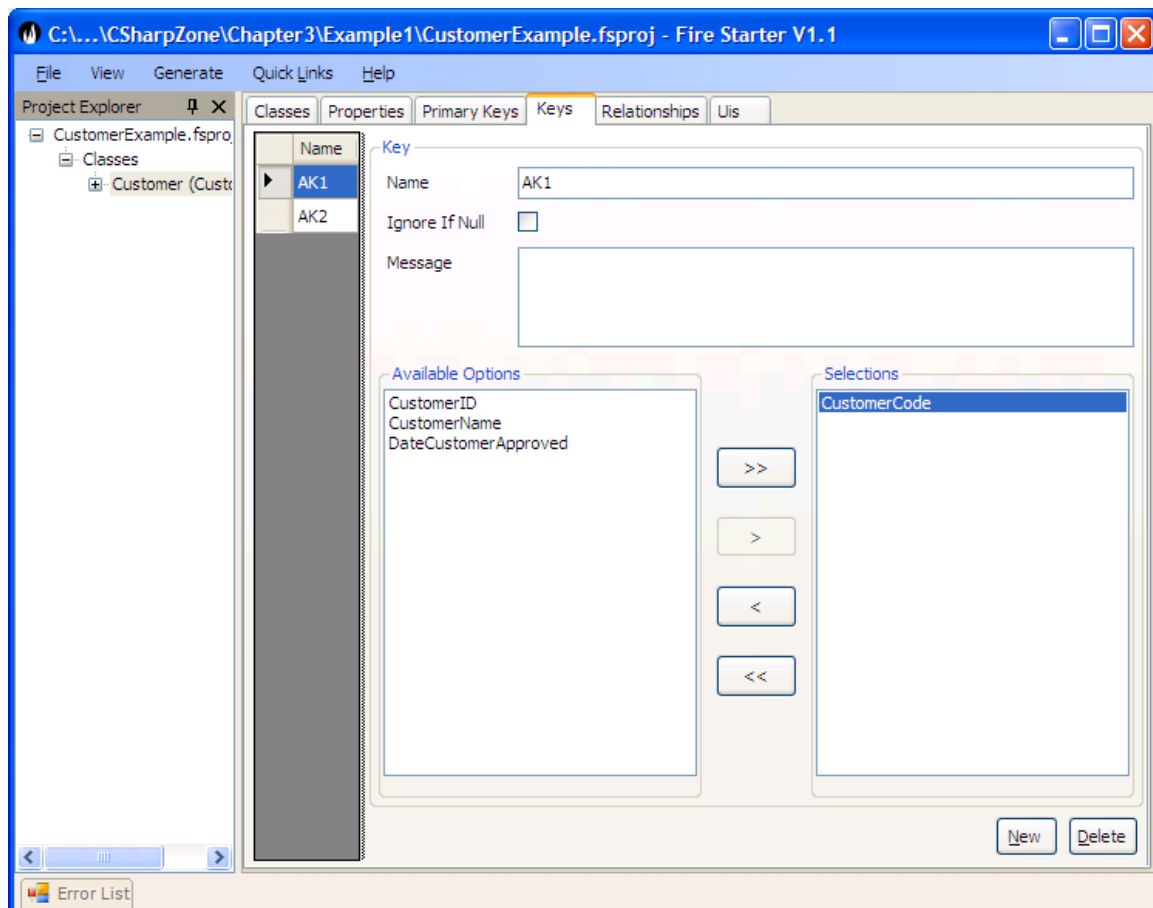
For restoring a new business object see Test_RestoreNewCustomer.

The more interesting test however is retrieving an object from the DataStore, editing it and then restoring it as shown in Test_RestoreLoadedCustomer

```
[Test]
public void Test_RestoreLoadedCustomer()
{
    //Tests Restoring a loaded business object.
    // This test shows that when the customer's Restore
    // method is called. The customers Status and Data is
    // restored so as to be the same as an object just loaded
    // from the DataStore.
    //-----Set up test pack-----
    Customer customer = CreateSavedCustomer();
    string origCustomerName = customer.CustomerName;
    customer.CustomerName = "New customer Name";
    //-----Assert Precondition-----
    Assert.IsTrue(customer.Status.IsDirty);
    Assert.AreNotEqual(origCustomerName, customer.CustomerName);
    //-----Execute Test -----
    customer.Restore();
    //-----Test Result -----
    Assert.IsFalse(customer.Status.IsDirty);
    Assert.AreEqual(origCustomerName, customer.CustomerName);
}
```

ALTERNATE KEY FOR A BUSINESS OBJECT

It is possible for a business object to have one or more alternate keys. An alternate key could be composite or simple. The alternate key identifies one or more properties which together acts as a rule preventing a duplicate business object from being persisted. A classical example of an alternate key for the customer is where the customer code must be unique (i.e. two customers with the same code cannot exist in the data store).



TODO: Reference test example code

IMPLEMENTING BUSINESS OBJECTS

Do something to explain this is advanced knowledge not required by most application developers.

When an application using Habanero is started, it loads the appropriate ClassDefs.xml (we will go into detail of where and when this happens in Section 5 – UI Layer). The ClassDefs are loaded into the class ClassDef which contains a static collection of the class definitions for each Business Object defined in the ClassDefs.xml.

CREATING A NEW BUSINESS OBJECT

When a business object of type Customer is created the business object is constructed with its BStatus appropriately set as new, not deleted and not dirty.

The business object properties and any related rules are retrieved for the Customer business object from the ClassDef class. From these class definitions a collection (BOPropCol) of Business Object properties (BOProp) is created for the customer. Each business object property tracks the current value for the Business object property, whether the property is dirty, valid etc. The BOProp also contains a reference to the definition for the Property i.e. The PropDef.

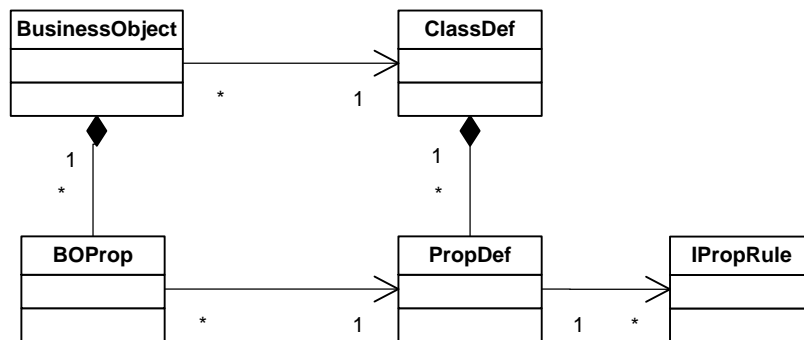


Figure 3.1: The relationship between a Business object instance and its definitions.

The above class diagram represents a pattern that is used throughout the framework and therefore warrants some discussion and explanation. The ClassDef and its related PropDefs are loaded from the ClassDefs.xml. The ClassDef and PropDef classes contain the definitions of a class and its properties for a particular BusinessObject type e.g. Customer. The definition classes ClassDef and PropDef contain the definitions that are common to all instances of the Customer. The BOProp contains only the data that is unique for a particular instance of the Business Object i.e. CustomerX (example with CustomerName, the PropDef defines that its type is a string and defines the PropertyRules associated with this property, while the BOProp for CustomerName will store the actual value for that instance of Customer).

SET PROPERTY VALUE

Let's demonstrate this with some tests by referring to the [TestSetCustomerBOProp](#) class. Note that the Application Developer would never use BOProp directly (The previous examples of setting and getting properties should always be used).

These tests are shown here purely to demonstrate how the framework implements Business Objects and their properties.

```
[Test]
public void Test_SetCustomerName_ToInvalidValidValue_FromValidValue()
{
    //When a property is set to an In Valid Value for a compulsory field
    // that has no value the broken rule is changed from compulsory to
    // Invalid Value.
    //-----Set up test pack-----
    Customer customer = new Customer();
    IBOProp customerNameBoProp = customer.Props["CustomerName"];
    customer.CustomerName = "Valid Name";

    //-----Assert Precondition-----
    Assert.AreEqual("Valid Name", customerNameBoProp.Value);
    Assert.IsTrue(customerNameBoProp.IsValid);
    Assert.AreEqual("", customerNameBoProp.InvalidReason);

    //-----Execute Test -----
    customer.CustomerName = "Inv";

    //-----Test Result -----
    Assert.AreEqual("Inv", customerNameBoProp.Value);
    Assert.IsFalse(customerNameBoProp.IsValid);
    StringAssert.Contains("'Inv' for property 'Customer Name' is not
        valid for the rule 'CustomerName'. The length cannot be less
        than 5 character", customerNameBoProp.InvalidReason);
}
```

From this test you can see that the BOProp tracks its current value (Value) as well as its Status (IsDirty, IsValid) and any invalid reasons. The Business Objects Status is then derived from the composite status of all its BOProps.

IMPLEMENTING PROPERTY RULES

Property Rules are implemented using a Strategy Pattern (See GOF). This pattern allows the application developer to easily extend the framework by adding new rule types and creating custom Property Rules for the application. (See Extending the Habanero Framework).

The property rules are implemented in such a way that they are available to the user interface allowing for the development of responsive applications with no duplication of rules.

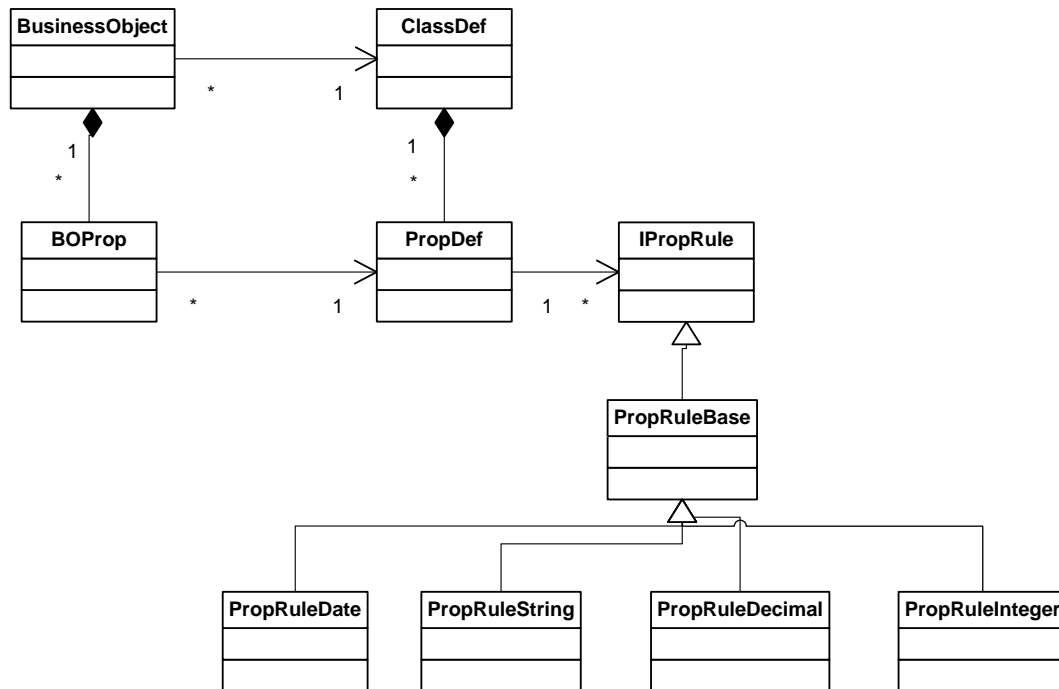


Figure 3.3: The relationship between the Business Object properties and its property rules.

IMPLEMENTING SAVING BUSINESS OBJECT TO DATASTORE

The details of how an object is saved to the data store are discussed in detail in Section 4. For the purposes of this chapter we are only going to look at how the Business Objects Properties are updated when the object is saved. This test demonstrates how the business object property is updated.

The BOProp has two values, namely the Value and the PersistedPropertyValue. The Value always holds the current value for that BOProp whereas the PersistedPropertyValue holds the value of the property that is currently held in the datastore. As illustrated in this test, a new Customer's BOProps have no PersistedPropertyValue, when the Business Object has been saved to the Datastore the PersistedPropertyValue is updated with the Value.

```

[Test]
public void Test_SaveBOProp()
{
    //Testing that when a customer is saved the Persisted Property
    // Value is updated to the PropertyValue
    //-----Set up test pack-----
    Customer customer = new Customer();
    const string newCustomerName = "Valid Name";
    customer.CustomerName = newCustomerName;
    customer.CustomerCode = "Code";
    IBOProp customerNameProp = customer.Props["CustomerName"];

    //-----Assert Precondition-----
    Assert.IsNull(customerNameProp.PersistedPropertyValue, "A new object
        should not have a persisted value");
    Assert.AreEqual(newCustomerName, customerNameProp.Value);

    //-----Execute Test -----
    customer.Save();

    //-----Test Result -----
    Assert.AreEqual(newCustomerName, customerNameProp.Value);
    Assert.AreEqual(newCustomerName,
        customerNameProp.PersistedPropertyValue,
        "After saving the PersistedPropertyValue should be backed up
        to the property value");
}

```

IMPLEMENTING BUSINESS OBJECT IDENTITY

The Business Object identity is implemented in a highly flexible yet simple manner in Habanero. The implementation follows the identity field pattern (Fowler - 216). The object identity (PrimaryKey) is defined as consisting of one or more properties (BOProp) of the business object. The ClassDef for the customer will be loaded with a PrimaryKeyDef. The PrimaryKeyDef will be loaded with the PropDef(s) that make up its primary key (in this case the CustomerID). When the Customer Object is created, the Customer Object is created with a BOPrimaryKey that references the CustomerID BOProp (i.e. the BOPrimaryKey is thus merely a collection of BOProps). If at any point the user requests the ID, the BOPrimaryKey Object will be returned.

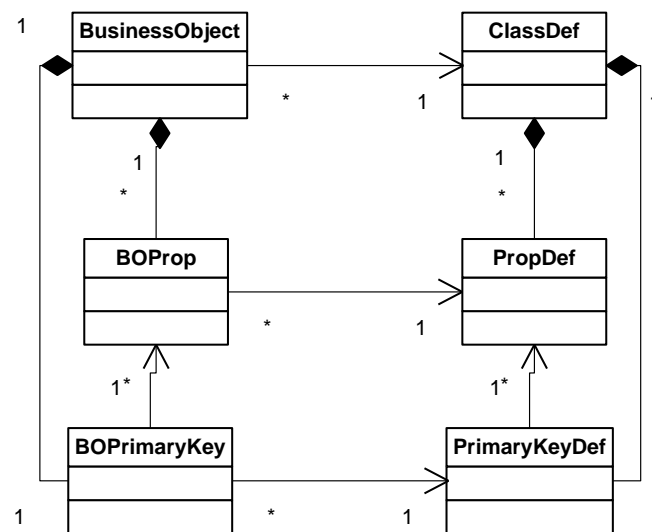


Figure 3.2: The relationship between a Business object instance and its primary key definition.

From this test it is evident that the PrimaryKeyDef is created with a single PropDef and that the BOPrimaryKey is created with a reference to the BOProp for CustomerID property. See the TestBusinessObjectIdentity class.

```

[Test]
public void Test_CreateCustomerWithIdentity()
{
    //This test shows the that the PrimaryKeyDef and BOPrimaryKey
    // are set up according to the class definition defined for
    // customer. It also shows that the CustomerID is automatically
    // set to a new GUID Value.

    //-----Execute Test -----
    Customer customer = new Customer();

    //-----Test Result -----
    ClassDef customerClassDef = customer.ClassDef;
    PrimaryKeyDef primaryKeyDef = customerClassDef.PrimaryKeyDef;
    Assert.AreEqual(1, primaryKeyDef.Count);

    IPrimaryKey customerPrimaryKey = customer.ID;
    Assert.AreEqual(1, customerPrimaryKey.Count);
    Assert.IsTrue(customerPrimaryKey.Contains("CustomerID"));
    IBOProp customerIDBOProp = customerPrimaryKey["CustomerID"];

    Assert.IsNotNull(customerIDBOProp, "Since the CustomerID is an object
        id it should be set to a value");
}

```


In the `TestBusinessObjectIdentity` class we also have the example `Test_CreateBO_WithNonGuidId` of creating an object that does not use a Guid Object ID. This Test is interesting because it also shows how a `BusinessObject` can be defined directly in code, without the use of `ClassDefs.xml`. We do not go into the details of defining `BusinessObjects` programmatically in the book – if you are interested look at the test and it's `BusinessObject`.

If you are using a non Guid Object identifier the following should be remembered:

- 1) All properties that are part of the `PrimaryKey` should be Compulsory and `WriteNew` (Properties cannot be null and cannot be changed once the business object has been persisted).
- 2) If you are using your own Number Generator to generate a unique identifier for the Business Object then you should refer to `Extending the Habanero Framework`.
- 3) If the properties must be mutable (i.e. changeable) then you should ensure that the modification of the ID can only be done via a separate business process which is well isolated and/or that only users who have a high level of authority can edit these properties. You should use concurrency control with pessimistic locking (see Section 4) when editing a primary key ensure that no concurrency control conflicts occur.

IMPLEMENTING ALTERNATE KEYS

Alternate keys are implemented in an almost identical manner to primary keys. In fact the `BOPrimaryKey` inherits from `BOKey` and `PrimaryKeyDef` inherits from `KeyDef`. The difference is in the relationship to the business object whereas a business object can only have one primary key it can have many alternate keys. As with the primary key the `BOKey` can be composite or single. The uniqueness of the alternate key is only verified when trying to persist the object to the database. The alternate key can be mutable or immutable as required by the business rules.

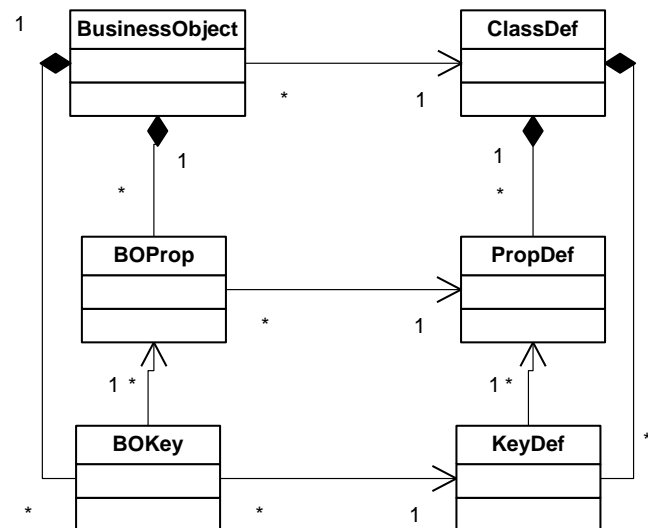


Figure 3.4: The relationship between a Business object instance and its alternate key definitions.

THE BUSINESS OBJECT MANAGER

It is essential when developing enterprise applications that a single business object is never represented by two instances of the same business object in the same application instance because of concurrency control issues. Imagine that a user loads one instance of Customer X via one user interface, then the same user loads a second instance of Customer X and edits the second instance of Customer X. If the user now returns to the original user interface, the edited data will not be shown i.e. the user will see the old data even though they just edited the object. If you want more details on this issue then refer to the [Identity Map Pattern - Fowler – 216](#).

In order to prevent the issue discussed above, the Habanero Framework implements the Identity Map Pattern using the `BusinessObjectManager` Class. The `BusinessObjectManager` is in essence a dictionary of Weak References to Business Objects. The weak references are keyed on the `ObjectID`. A weak reference is in essence a safe pointer to an object i.e. it is a reference to an object that does not prevent the object from being garbage collected.

There are no tests in `CustomerExample` for the `BusinessObjectManager` because most of its methods are internal but the `Habanero.Test.BO` project (which ships with the framework) contains the `TestBusinessObjectManager` Class that shows the behaviour of the object manager in detail.

In summary:

- When a new Business Object is first persisted it is added to the BusinessObjectManager.
- When a Business Object is retrieved (loaded from the datastore) for the first time, it is added to the BusinessObjectManager.
- When a Business Object that already exists in the BusinessObjectManager is retrieved, the already existing Business Object from the BusinessObjectManager is returned. (Note: there are various options for refreshing or not refreshing the already loaded business object that are covered in Section 4 – Data Access Layer).
- When a Business Object is collected by the garbage collector it is removed from the BusinessObjectManager.

LOADING BUSINESS OBJECT FROM DATASTORE

The details of mapping and loading a business object from a data store will be discussed in Section 4.

For this section we will look at how the BOProp is loaded with the appropriate data. When a Business Object is loaded from the datastore the BOProp's PersistedPropertyValue and Value are loaded with the value currently held in the DataStore.

TODO: reference test

SUMMARISING THE BUSINESS OBJECT DEFINITION

In summary the business object consists of a BOPrimaryKey and a collection of BOProperties.

The BOProperties are responsible for maintaining their own state and for validating any business rules such as ReadWriteRules, Compulsory rules or IPropRules.

The Business Object is constructed with the appropriate BOPrimaryKey and BOProps based on the ClassDefinition contained in the ClassDefs Collection.

The BOProp is validated using the Property Rules associated with its PropDef.

The advantages of having a Business Object made up of intelligent BOProps representing the Business Object's data instead of simple fields should already be evident. The advantages will become even clearer when studying the UI Layer and the DataAccess layer. In preparation however we will briefly mention the following:

- 1) The BOProp has the knowledge of whether the user is allowed to edit it. The appropriate user control can therefore be enabled or disabled appropriately.
- 2) The BOProp has direct access to its property rules. The user interface can utilise these rules directly to do data input validation. This results in highly responsive user interfaces while still ensuring that the Business Objects enforce all their rules with no duplication of rules.
- 3) The PropDef contains the required information on how it is mapped to any field in the DataStore allowing the application developer to load Business Objects without having to know the database table or field names.

APPLYING SECURITY TO BUSINESS OBJECTS

Every enterprise application requires reliable application security. Security is a broad topic and encompasses many areas that beyond the scope of this book. However, having said that, it is a requirement that security is applied appropriately to the Business Object Layer. Security based rules for accessing, deleting and editing data is just as appropriate in the Business Object Layer as ReadWrite rules, Compulsory rules and Validation rules. In other words authorisation rules regarding which functions and data a user can access logically belong in the Business Object layer and the Business Objects should implement them where appropriate.

Security is one of the areas of application development that involves many decisions and is highly dependent on the environment in which the application is being deployed. It is for this reason, that we do not include a security policy in the business object layer but instead provide the strategy (and thus the hooks) that enables an application developer to implement his/her own security policy.

To understand security we need to clearly differentiate between authentication and authorisation. Authentication involves the verification that the user is who he/she says he/she is. Authorisation involves determining whether the user is authorised to use a particular function (whether it is an application, a business object, a business object property, a user interface or a report).

In implementing authentication, there are primarily two security options:

- 1) Integrated Authentication: Use Windows Integrated security i.e. use the Windows user who is currently logged onto the Windows operating system as the authenticated user.
- 2) Custom Authentication: A custom application is used for logging users into the application or into a set of applications. The logged on user as per the application could be different from the logged on user as per the Windows operating system.

The Habanero Framework supports the use of both Integrated authentication and custom authentication. Authentication will be discussed in more detail under Section 5 – The UI Layer.

In implementing authorisation, there are once again two main options:

- 1) Integrated Authorisation: Use Windows Integrated Authorisation. **Windows AD, for instance**, allows the user to be set up as a member of certain roles or user profiles. These profiles can then be used by the application to grant or deny the user permissions for specific functions. If integrated authorisation is used then Integrated authentication must be used.
- 2) Custom Authorisation: A custom application is used to associate the user as a member of a set of Roles/Profiles. These profiles are then used by the application to grant or deny the user with permission for certain functions. Note that a user could be authenticated using Windows authentication but the profiles could be managed using custom authentication.

The Habanero Framework supports the use of both Integrate Authorisation and Custom Authorisation.

IMPLEMENTING AUTHENTICATION

Authentication will be discussed in more detail under Section 5 – The UI Layer.

IMPLEMENTING AUTHORISATION

The Business Object layer has three areas in which security rules are typically applied in. These are:

- 1) Implementing CRUD (Create, Read, Update, and Delete) rules for a Business Object.

Typically a certain Group (Role/Profile) of users may have different permissions to Creating, Reading, Updating and Deleting Business Objects (Typically called CRUD permissions).

- 2) Implementing Read Write Rules for a particular property.

The common use is that some users may be able to view/edit certain Properties of a Business Object and others users may be able to view/edit other properties.

- 3) Implementing Permission on specific methods.

Once again specific methods may only be executed by certain groups of users.

The security rules are implemented by the business object layer. With this approach the user interface can respond appropriately to the rules by enabling or disabling controls without requiring duplication of the rules in the user interface. This allows the business objects to be safely used by different clients (e.g. Windows Forms, ASP, WebServices or other Services).

On a practical basis the Business Objects are by default distributed in a component that is separate to the User Interface. If security is only implemented in the User Interface a malicious user could create an application that uses the Business Logic Layer DLL's and access functionality that is not normally allowed.

Before we go any further we will look briefly at the DotNet Security Model. This is a role-based Security Model available in the System.Security.Permissions namespace.

Security can easily be implemented on Methods using the following Code. This enforces that only a user who is a member of the accounts role can access this method.

```
[PrincipalPermission (SecurityAction.Demand, Role="Accounts" ) ]  
  
public long ApproveOrder() { //DoSomething }
```

By Default the Role="Accounts" is set up in a User Group in Windows, AD etc. This is seldom useful for serious Application Development because the UserGroup Roles are seldom finely grained enough for an Enterprise Application.

For this book we are going to focus on how to use the appropriate Security Components to implement Security in the Business Object Layer.

This discussion therefore assumes that you have the role based security components set up appropriately. It turns out that this is the simple part – the next thing to achieve is to assign permissions to any relevant entity based on the role. For instance, a member of the Accounts Group can create an Invoice and edit an Invoice but cannot delete an invoice or edit an Invoice's Text. A member of the Account Managers Group can edit an Invoice and can edit an Invoice's Text but cannot delete an Invoice.

Role based security

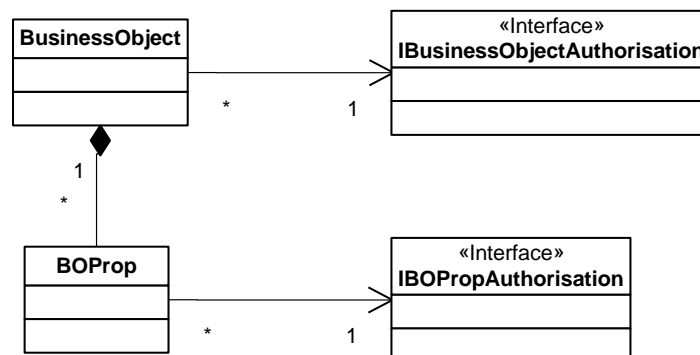
The Dot Net security model provides the user with a powerful role based model for implementing security. The user can be assigned roles by windows e.g. by AD or the user can be assigned roles by custom code. We have seldom found that the Roles and groups assigned to users for network permissions are fine grained enough for application security so we have tended to use windows authentication with custom roles. (see the Section 5 – UI Layer).

APPLYING AUTHORISATION TO A BUSINESS OBJECT

Once roles are assigned to a user, the framework still needs to assign the user with permissions to implement specific functionality e.g. a member of the 'Account Manager' role may edit the description on a non revenue recognised invoice.

In our experience the authorisation requirements are so varied that the only practical method to implement security is using the strategy pattern. Each business object has a reference to an `IBusinessObjectAuthorisation` object. This allows us to deal with complex security policies e.g. scenarios where a Business Object such as an invoice could have the description edited by members of the Customer Services and by member of Account Manager. Once the Invoice had been printed and sent to the tax authorities for a clearance certificate then only the Account Manager could edit the description. The design illustrated below allows this flexibility in implementing business objects security. In most cases the same authorisation object will be applied to all business objects of a certain type. In this case the business objects will all reference the same instance using the singleton pattern.

As can be seen in the diagram below a business object references an instance of the `IBusinessObjectAuthorisation` class.



Chapter 3-5: The relationship between Business Object and the security interface.

The tests and setting up of tests for security is somewhat complicated so we will not go into all the details in this book. The test class `TestSecurityCustomer` has all the details required.

In summary:

if the security policy does not allow the business object to be read then the `customer.IsReadable(out message)`; will return false and the message will give the reason that it is not readable.

The `IsEditable`, `IsDeletable` and `IsCreatable` work the same.

IsCreatable is an instance method and not a static method because static methods cannot be overridden. These methods can be used by the user interface enable or disable controls. In addition, these methods are used by the business object itself to prevent the Creating, Reading, Editing and Deleting of the Business Object. The tests in TestSecurityCustomer illustrate this.

Below we have shown only the tests for Reading but there are similar tests for Editing, Deleting and Creating.

```
[Test]
public void Test_BusinessObjectAuthorisation_AllowRead_False()
{
    //-----Set up test pack-----
    IBusinessObjectAuthorisation authorisationStub = new
        AuthorisationStub();
    Customer customer = new Customer();
    customer.SetAuthorisation(authorisationStub);
    //-----Assert Precondition-----
    Assert.IsFalse(authorisationStub.IsAuthorised(
        BusinessObjectActions.CanRead));
    //-----Execute Test -----
    string message;
    bool isReadable = customer.IsReadable(out message);
    //-----Test Result -----
    Assert.IsFalse(isReadable);
    StringAssert.Contains("The logged on user", message);
    StringAssert.Contains("is not authorised to read ", message);
}

[Test]
public void Test_LoadExistingBO_Fail_AllowRead_False()
{
    //-----Set up test pack-----
    IBusinessObjectAuthorisation authorisationStub =
        GetAuthorisationStub_CanCreate_True();
    Customer customer = CreateNewCustomerValid();
    customer.SetAuthorisation(authorisationStub);
    customer.Save();
    authorisationStub = GetAuthorisationStub_CanRead_False();
    customer.SetAuthorisation(authorisationStub);
    IPrimaryKey id = customer.ID;
    BusinessObjectManager.Instance.ClearLoadedObjects();
    //-----Assert Precondition-----
    Assert.IsFalse(authorisationStub.IsAuthorised(
        BusinessObjectActions.CanRead));
    Assert.IsFalse(customer.Status.IsNew);
    //-----Execute Test -----
    customer =
        BOREgistry.DataAccessor.BusinessObjectLoader.GetBusinessObject
        <Customer>(id);
    try
    {
        customer.GetPropertyvalue("Prop1");
        Assert.Fail("expected Err");
    }
    //-----Test Result -----
    catch (BusObjReadException ex)
    {
        StringAssert.Contains("The logged on user", ex.Message);
        StringAssert.Contains("is not authorised to read ", ex.Message);
    }
}
```

From these tests it is evident that all you have to do is set the correct authorisationpolicy when the Business Object is being constructed.

```
protected internal Customer(ClassDef def) : base(def)
{
    SetAuthorisationRules(new CustomerAuthorisationRules());
}
```

Once this is done, the framework will do everything else that is required. The example project [??? shows this in great detail](#). If no authorisation rules are set for a business object then it is assumed that everyone has create, read, update and delete permissions.

Note the `IsEditable`, `IsReadable` etc methods are overridable in the Business Object. It is important that these methods should be marked as `Sealed` methods in the Sub Classes since this will prevent malicious use of the Component by a developer who might try to inherit from the base class.

APPLYING AUTHORISATION TO A BUSINESS OBJECT PROPERTIES

Business object properties follow a similar pattern to the business object with the difference being that a `BOProp` can only have Read and Write (Edit) permissions set. Setting permissions on a particular Business Object Property (`BOProp`) are much less frequent than setting permissions on a Business Object.

The `BOProp` Rules are again setup for a particular Business Object in [its constructor redo should be set from a lazy initialisation method e.g.](#)

```
protected internal Customer(ClassDef def) : base(def)
{
    IBOPPropAuthorisation propAuthorisationStub = new
        CustomerNameAuthorisationPolicy();
    BOProp propCustomerName = (BOProp)this.Props["CustomerName"];
    propCustomerName.SetAuthorisationRules(propAuthorisationStub);
}
```

The `BOProp` has methods `IsEditable` and `IsReadable`. These methods return true or false based not only on the security policy for the `BOProp` but also on the ReadWrite Rules set for the `BOProp`. These two methods can be used by the user interface to enable/disable a control or in the case of `IsReadable` being false the user interface can blank out the control to show that it is not readable. Regardless of the user interface, the Business Object and its business object properties will prevent the reading and writing of business objects properties where the rules prevent it. For detailed tests showing this functionality refer to the `TestBOPPropAuthorisation` class in the `Habanero.Test.BO.Security` namespace of the `Habanero.Test.BO` Project that is downloaded with the application framework.

APPLYING AUTHORISATION TO A BUSINESS OBJECT METHOD

The business object methods can also have security applied to them. In these cases the mechanism is similar but different because there is only one action that the user can do on a method and that is to execute the method.

The Interface used is `IBOMethodAuthorisation`. An authorisation rules object of this type is created. The roles that can execute the method are added using `AddAuthorisedRole` probably during object construction and the method then calls `IsAuthorised` on the authorisation rules object prior to executing. If the use is not authorised, then the appropriate error must be raised by the method.

EXTENDING THE BUSINESS OBJECT

There are a number of points provided in the Business Object framework that are specifically intended for extending the framework. We will briefly mention each method, how it is intended to be used and where you can find tests that illustrate their use.

AFTERLOAD

This method is virtual on the Business Object and by default, implements no code. The application developer can override this method to carry out any custom code that is required after the object is loaded from the DataStore. This method will be called if the object is loaded for the first time or if the object is refreshed.

AFTERSAVE

This method is virtual on the Business Object and by default implements no code. The application developer can override this method to carry out any custom code that is required after the object is saved to the DataStore. This method will be called every time the object is persisted.

UPDATEOBJECTBEFOREPERSISTING

This method is virtual on the Business Object and by default implements code that adds a transaction log object to the transaction committer if a transaction log is defined for the Business object. The application developer can use this method to carry out any custom code that is required before the object is persisted to the DataStore. This method is called just before the object is persisted and allows the application developer to implement any custom code. A typical example where we use this is where there is a requirement to generate a custom number or sequential code for a business object, e.g. each invoice must have a number which is unique. For these cases, a particular implementation of the `INumberGenerator` class is usually used (note the framework contains a few implementation of `INumberGenerator`). The `INumberGenerator` object is then added to the transaction Committer to ensure that the number generator and business object are both either updated or rolled back. For more details on the TransactionCommitter see Section 4 – Data Access Layer.

ADDING RULES FOR A BOPROP

The framework allows the Application developer to add custom BOPROP Rules in addition to the standard rules defined in the `ClassDefs.xml`.

The PropRules are added to the BusinessObject in its constructor e.g.

```
protected internal Customer(ClassDef def)
    : base(def)
{
    BOPROP propCustomerName = (BOPROP)this.Props["CustomerName"];
    PropDef propDef = (PropDef) propCustomerName.PropDef;
    propDef.AddPropRule(new MyPropRule(this, propCustomerName));
}
```

ARE CUSTOM RULES VALID?

This method is virtual on the Business Object and by default returns true. This method is called prior to saving any Business Object. The application developer can override this method so as to implement any complex custom rules that cannot be handled by the IPropRule strategy. The disadvantage of placing custom rules in this method is that they are unavailable as broken rules to the user interface, thus making it difficult to show them using the ErrorProvider. Most rules could be implemented using either the IPropRule Strategy or the AreCustomRules Valid strategy the reason we allow both is that rules where the value of 1 property relies on the value of another property may not be best validated using PropRules. The best way to demonstrate this is via an example

Lets assume you have a Business Object that has a property for Date Of Birth and also for Identity Number. The identity number has 4 numerics that indicate the month and year of birth. There is a requirement that the date of birth is validated against the Identity number to ensure the month and year of birth are correct. A prop rule could be set on the DateOfBirth Validating against the Identity Number but this would imply that the identity number has already been captured causing a coupling between the Business Object being updated with an identity number and the Date of birth.IsEditable

By default the IsEditable Method checks the authorisation Rules set up for the Business Object. The Application developer can however override this and implement any other rules that may change whether the object is edited or not.

```
public virtual bool IsEditable(out string message)
{
    message = "";
    if (_authorisationRules == null) return true;
    if
        (_authorisationRules.IsAuthorised(BusinessObjectActions.CanUpdate)) return true;
    message = string.Format("The logged on user {0} is not authorised to update {1} Identified By {2}",
        Thread.CurrentPrincipal.Identity.Name, this.ClassName,
        this.ID.GetObjectId());
    return false;
}
```

ISDELETABLE

By default the IsDeletable Method checks the authorisation Rules set up for the Business Object. The Application developer can however override this and implement any other rules that may change whether the object is deletable or not.

LOGGING EDITS

The Framework once again provides the application developer with the hooks to implement a logging strategy if he/she requires. The required logging strategy is set up for the Business Object in its constructor using the `protected void SetTransactionLog(ITransactionLog transactionLog)` method. The framework also ships with objects that implement a particular logging strategy. For more details see the `TestTransactionLogger` class which shows the use of the `TransactionLogTable`.

CONCURRENCY CONTROL

The framework once again provides the application developer with the hooks to implement a any concurrency control strategy if he/she requires. Concurrency Control is a complicated topic and requires a fair amount of interaction with the database to implement. The subject is therefore fully discussed in the Section 4 – The Data Access layer. The Habanero Framework ships with a number of different implementations of various concurrency control strategies. The test classes

`TestConcurrencyControl_OptimisticLockingVersionNumberDB` and `TestConcurrencyControl_PessimisticLockingDB` provide examples of setting up and using concurrency control in a business object.

CHAPTER 4 DOMAIN MODEL -THE BUSINESS OBJECT COLLECTION

Business Objects are frequently loaded and used in collections. The common practice is to load a set of business objects based on some search criteria or to load all business objects. The business object collection can then be used for calculations, to display in a grid, list or report.

LOADING A BUSINESS OBJECT COLLECTION

We will start with the simple loading a collection of Business Objects.

Business Object Collections are loaded in a very similar manner to Individual Business Object i.e. you can use either a string criteria or a criteria object.

The tests for loading Business Object Collections are contained in the `Test_LoadCustomerUsingStringCriteria` Class. We are not going to repeat all the code in the book but essentially the code for loading a business object collection is:

```
const string loadCriteria = "CustomerCode Like Code%";
IBusinessObjectCollection loadedCustomers =
    GetBusinessObjectLoader().GetBusinessObjectCollection<Customer>(loadCriteria);
```

The Criteria Objects and Criteria Strings are much more powerful than shown so far and include all the normal operators required i.e. AND, OR, opening '(', closing brackets ')', <>, =, <=>, >=, <=, >, <, IS NOT NULL, IS NULL, NOT LIKE, LIKE. All these operators work for loading collections of objects as well as for loading individual objects.

Note however that if you try loading an individual object with a Criteria and more than one object is found in the database an error will be raised.

This test provides an example of how complex loadCriteria can be built up using strings. You can also build

```
public void Test_LoadCustomerUsingCriteriaString_Complex()
{
    ///This test shows that if a persisted object is loaded from the
    /// dataStore using the BusinessObjectLoader.GetBusinessObject.
    /// Then an object with the exact same status and data as
    /// the persisted object is loaded.
    //-----Set up test pack-----
    Customer customer = CreateSavedCustomer();
    Customer customer2 = CreateSavedCustomer();
    Customer customerNoMatch = CreateSavedCustomer("CustName",
        "NotMatch");
    Customer customerNoMatch2 = CreateSavedCustomer("CustName", "Code");
    Customer customerMatchOnCustomerName =
```

complex Criteria using Criteria objects.

The business Object collection also provides helper methods that wrap the Business Object loader and provide the same functionality for loading a collection of business objects. These methods are `customers.LoadAll()`, various overloads of `customers.Load()` and various overloads of `customers.LoadWithLimits()` See `Test_LoadCustomer_UsingCollectionLoad`.

Loading also provides a number of other capabilities including loading only a selected number of records (`customers.LoadWithLimits()`) and loading in a specific order (i.e. `OrderBy` Criteria). Once again the `orderBy` is always property names of the business objects and never database field names. See `Test_LoadCustomer_UsingCollectionLoad_LoadWithLimit`.

For a full set of tests showing all the functionality for Loading Business Object Collections see `TestBusinessObjectLoader_GetBusinessObjectCollection` in the `Habanero.Test.BO` project.

Future must be able to build custom criteria for any object that just need to put values e.g Criteria object called search by Surname, firstname and date of birth. All the UI developer needs to do then is use these criteria objects with the load. This will significantly improve the UIDevelopers experience especially with complex Criteria. see marks stuff

Load record 2-10 instead of only loadwithlimit

In summary you can load a collection of Business Objects in a number of ways always referring only to the business object properties thus totally isolating the application developer from the database implementation.

MAKING EDITS TO A BUSINESS OBJECT COLLECTION

In addition to loading a collection the developer can manipulate the collection in a number of ways. The principles of how Habanero manages editing a Business Object Collection are similar to editing a Business Object. To achieve this functionality the Business Object collection maintains 4 collections namely

- The `CreatedBusinessObjects`, a list of business objects created by the collection.
- The `RemovedBusinessObjects`, a list of business objects removed from the collection.
- The `PersistedBusinessObjects`, a list of business objects representing the list of business objects as per the last time the collection loaded from the database or persisted to the database.
- The `MarkedForDeleteBusinessObjects`, a list of all business objects marked as deleted by the collection.
- The primary collection, a collection showing the `PersistedBusinessObjects` plus the `CreatedBusinessObjects` Less the `RemovedBusinessObjects` and `MarkedForDeleteBusinessObjects`.

CREATING BUSINESS OBJECT

The business object collection can create a business object (`customers.CreateBusinessObject()`). The business object of the appropriate type will be created and added to the business objects internal collection of created business objects (`customers.CreatedBusinessObjects`). The Business Object will also be added to the current collection and will thus be available for viewing/editing in grids, lists etc.

If the collection is saved then these created business objects will be saved (`customers.SaveAll()` and `customers.SaveAllInTransaction(transaction)`) to the database.

(create tests and reference)

If the created Business Object is saved independently it will be removed from the created list and added to the Persisted list.

If the collection is restored (all edits cancelled. `Customers.RestoreAll`) then the collection will be reverted to its original state i.e. the created Business Objects will be cleared and the current collection will be restored from the PersistedBusinessObjects.

(create tests and reference)

ADD A BUSINESS OBJECT

The developer can add an existing business object to a business object collection.

If the business object is new (`customer.Status.IsNew`) then the business object will be added to the CreatedBusinessObjects list as well as the primary list. (see test??).

If the business object is persisted (`customer.Status.IsNew`) then the business object will be added to the Current collection only. (see test??).

REMOVE A BUSINESS OBJECT

The developer can also remove a business object from a business object collection (`Remove()` and `RemoveAt()`).

If the business object is new then it is removed from the current collection and from the CreatedBusinessObjects list. See test

If the business object is not new then it is removed from the current list and added to the RemovedBusinessObjects list See test

MARKFORDELETE A BUSINESS OBJECT

The developer can also mark a Business Objects for deletion from a business object collection.

If the business object is new then it is removed from the current collection and from the CreatedBusinessObjects list and MarkedForDeletion. See test

If the business object is not new then it is removed from the current list, marked for deletion and added to the DeletedBusinessObjects list See test

PERSISTING BUSINESS OBJECT COLLECTIONS

A business object collection can be persisted via the .SaveAll method. All the added, removed, created and deleted business objects will be persisted and their collections cleared. [See test](#)

REFRESHING A BUSINESS OBJECT COLLECTION

A business object collection can be refreshed from the database. When refreshing a collection the collection will be reloaded.

Any dirty business objects will not be refreshed. Any Business Objects in the Removed or deleted list will not be shown in the Current list. Any items in the Created List [and Added](#) will be shown in the current list. [See test](#)

RESTORE (CANCEL EDITS) A BUSINESS OBJECT COLLECTION

As with a business object a business object collection can be restored. The business object collection will be restored to its persisted stated. The created, deleted and removed lists will be cleared. (i.e. the items in the persisted list and the current collection will be identical). [See test](#)

CLEAR A BUSINESS OBJECT COLLECTION

This clears the current collection, persisted, removed, deleted and created list. [See test](#)

OTHER BUSINESS OBJECT COLLECTION FUNCTIONS

The business object has other methods Sorting, Intersection and Union, Insert.

[\(TODO: Insert must work the same as add in terms of inserting a new object\)](#)

CHAPTER 5 – RELATIONSHIPS

OVERVIEW

The next logical step when using domain modelling is to model relationships between the Entities in our model.

A CODE EXAMPLE

Consider how this might be useful when writing code: let's say that an Invoice can have zero or more Invoice lines, and we wish to calculate the total value of the invoice by totalling the value of each line. We could put a

```
public property decimal TotalValue {
    get {
        decimal total = 0.0;
        this.InvoiceLines.ForEach(line => total += line.TotalValue);
        return total;
    }
}
```

property on the Invoice class such as:

The Invoice class now has the ability to give us its total value, and this code is made simple by a relationship that is modelled between `Invoice` and `InvoiceLine` (giving us the `InvoiceLines` property on the `Invoice` class).

In a similar fashion, given an `InvoiceLine` we might want to find out some detail about the product that it applies to, such as the description of it to print on the invoice.

```
String lineDescription = invoiceLine.Product.Description;
```

Here we have modelled a relationship between `InvoiceLine` and `Product` – a “single” relationship, meaning there is only one `Product` for a given `InvoiceLine`, and thus the `Product` property returns the actual `Product` object which has the `Description` property on it.

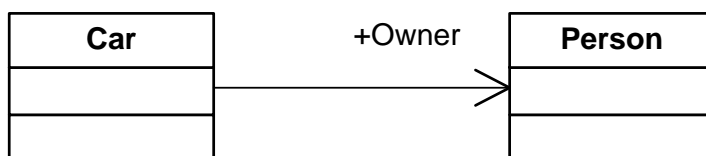
It’s immediately obvious how these constructs make logic code more intuitive as the code becomes more expressive and a closer match to the domain.

RELATIONSHIP CONCEPTS

Relationships can always be expressed in a sentence, and, in fact, if this sentence does not make sense it offers an indication that the relationship does not fit in the Ubiquitous Language of your model. For instance, consider the sentence:

A car is owned by a person.

This denotes a relationship between a car and a person, where the context is that of ownership. This can be modelled in UML as follows:



Note that there can be more than one relationship between two entities. For example, a car can be owned by one person while being driven by another. You can also have an association between an entity and another entity of the same type – for example a person could have siblings (who are also people).

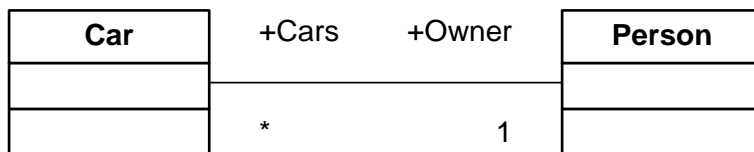
DIRECTIONALITY

The sentence above indicates that there is *directionality* in the relationship, as not much can be inferred in terms of the relationship in the other direction (for instance, there is no indication whether more than one car can be owned by a person. We can’t assume things from the real world when doing models otherwise we’d end up with a model bigger than necessary). A relationship that can only be traversed in one direction is *unidirectional*.

A fuller description of the relationship might read:

A car is owned by a person. A person owns zero or more cars.

This leads to a model like this:



This time there is no *directionality*. In other words the relationship is *bi-directional* (note that it could have been modelled as bi-directional before, but if had done that we would have been implying things that were not explicit in the relationship description). For bi-directional relationships it is conventional to drop the arrowheads in keeping with the agile approach (leave unnecessary things out.)

MULTIPLICITY

We also know the *multiplicity* of the relationship now. A car can, at any point in time, be owned by one, and only one person (this again suggests that we are not modelling the real world in general, only the scope of the system, and in the system a car always has one owner.) A person, on the other hand, can own zero, one, or any number of cars.

OPTIONALITY

One more concept when modelling relationships is *optionality*. In our model, because the description says a car must be owned (by not giving the option of zero owners), this is not an optional relationship – a car must have an owner. This is denoted by the number 1 next to the person end of the relationship in the diagram. If this was optional the multiplicity be 0..1. The second half of the relationship description indicates that a person can have zero or more cars, so this is an optional link as denoted by the * (or sometimes by 0..*). If a person always owned at least one car the multiplicity indicator would be 1..*.

TRANSITORY AND PERSISTENT

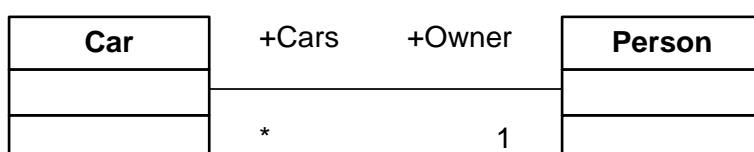
Relationships, as defined in Habanero and modelled in FireStarter are necessarily *persistent*. The domain model usually does not contain *transitory* relationships, that is, relationships between objects that are not persisted, so this discussion is purely centred on persistent relationships.

RELATIONSHIP TYPES

ASSOCIATION

The simplest relationship type is the *association*, which denotes a loose association between two Entities. The above example is repeated here for convenience:

A car is owned by a person. A person owns zero or more cars.

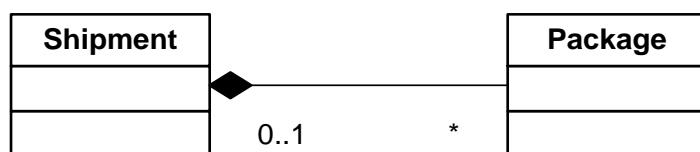


The example given above between the car and the person is an association: each of the Entities can exist in their own right and the relationship simply links them in some way. A car can then later be delinked from one owner (person) and attached to a different owner. A person, in turn, can have all their cars removed and be left carless, or receive new cars.

AGGREGATION

Often objects are made up of other objects, or need to be treated as a collective. For example, a delivery company might bag a bunch of parcels together into a shipment to go to a certain country – in this case it would make sense that the shipment is made up of one or more parcels, and that a parcel is part of a shipment. This type of relationship is referred to as an *aggregation*.

A shipment consists of one or more packages. A package is either unassigned or assigned to one shipment



Note that while a shipment consists of one or more packages (and a package is thus a part of a shipment), a package can exist without a shipment before it is assigned to a shipment, and can be moved from one shipment to another. This type of qualification shows that indicates that this is an aggregation as opposed to a composition.

COMPOSITION

When an object is composed of other objects, or it is wholly responsible for interaction with its part objects you have a strong form of aggregation called *composition*. An example is that of an Invoice and its Invoice Lines – an invoice line cannot exist outside of being part of an invoice, and all outside interaction should be done through the Invoice.

An invoice consists of invoice lines. An invoice line cannot exist other than on an invoice.



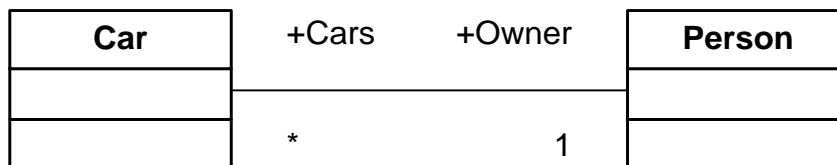
Composition is sometimes referred to as *strong aggregation* and the two are denoted using the same diagram representation in UML 2.0. The difference between them is subtle, but important. In our examples it is evident that a package can be searched for and interacted with (in a system) on its own. A tracking number, for example might be used to search for the package, check its current status and see what shipment that package is on. In other words, the package has *identity of its own* apart from the shipment that it's on (indicating aggregation). An invoice line on the other hand only exists as a constituent part of an invoice and does not have identity apart from the invoice that it is on (indicating composition, or strong aggregation).

MANY TO MANY RELATIONSHIPS

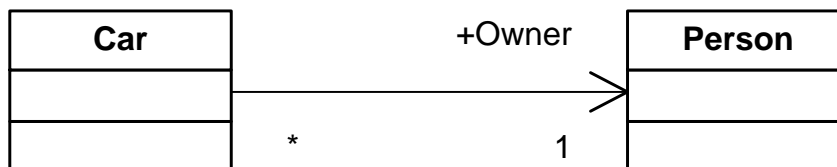
USING RELATIONSHIPS IN HABANERO

MODELLING RELATIONSHIPS

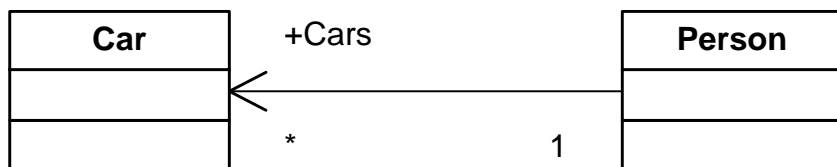
All relationships in Habanero are modelled as uni-directional relationships. This means that a bi-directional association such as:



Is modelled in Habanero as two uni-directional associations:



1. A car is owned by a person



2. A person owns zero or more cars.

The net effect is exactly the same model, as together the two relationships add up to the original relationship description.

The first, the relationship from Car to Person, will be depicted in code as a property called Owner on the Car class. This property returns an object of type Person:

```
public Person Owner { get; }
```

This relationship is what Habanero calls a *single* relationship. These can have a multiplicity zero or one and always return either null or a single object.

The second, the relationship from Person to Car will be depicted in code as a property on the Person class called Cars (this denotes ownership clearly enough, so I'm not going to qualify the name as OwnedCars or something similar). Cars returns a `BusinessObjectCollection<Car>`, a collection of Car objects:

```
public BusinessObjectCollection<Car> Cars { get; }
```

This relationship is what Habanero calls a *multiple* relationship. These can have a multiplicity of any number, and always return a collection.

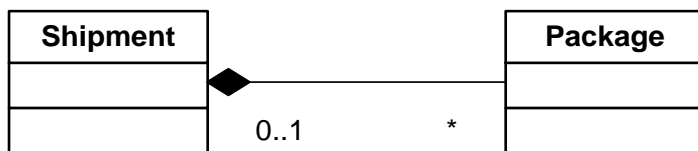
Thus a many-to-one relationship such as the Car/Owner relationship will consist of one single relationship (Car to Person, called Owner) and one multiple relationship (Person to Car, called Cars).

In Habanero these two relationships, Owner and Cars, are considered *reverse relationships* of each other. When traversing the Owner relationship from Car to Person, the reverse relationship is the Cars relationship. This is important to model as without the link between the two relationships Habanero may not treat them as a proper bi-directional relationship. Linking the two uni-directional relationships makes them behave as one bi-directional relationship.

If the relationship you are modelling is a uni-directional relationship then no reverse relationship will be required.

MODELLING AGGREGATION AND COMPOSITION

The above example involved a pure association relationship. If we take an aggregation or composition relationship, such as our Shipment/Package example, the situation is only slightly different.



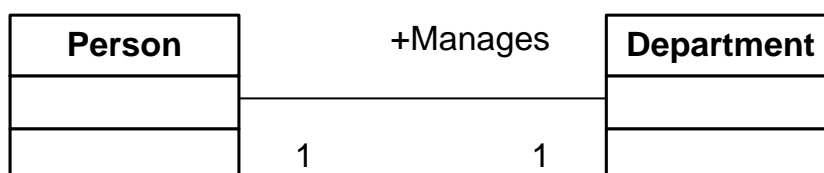
When breaking this relationship down into two uni-directional relationships for Habanero, the relationship from Shipment to Package would be modelled as aggregation as normal, but the relationship from Package to Shipment must be modelled as an association (or not at all if you don't ever need to traverse from a Package to its Shipment). If both are modelled as aggregations Habanero will raise an error when they are used. The diagram is in fact indicating this because the aggregation icon is only connected to Shipment, not to Package as well.

The same thing applies to composition – only one of the two directions must be modelled as composition, the other must always be an association.

ONE TO ONE

Modelling a one to one relationship is similar.

A manager manages a department. A department has one current manager.



Two uni-directional single relationships that are linked correctly will model this relationship. There is an extra consideration to note – it is important that only one of these relationships is configured to be the one that

owns the foreign key. If both are considered to have the foreign key then Habanero will throw an error when the manager of a department is set.

RELATIONSHIP COLLECTION BEHAVIOUR

In Habanero, the first choice to make when modelling is whether to use a single or a multiple relationships. This is not the only item required in order to fully describe the multiplicity of the relationship, but is the most fundamental of the properties of a relationship as it determines whether it points to a single object or a collection of objects, affecting the resultant client code.

Taking the above Owner/Car relationship as an example, the Owner relationship is modelled as a Single relationship, while the Cars relationship is a Multiple relationship (hence the Owner property returning a single Person object and the Cars property returning a collection of Car objects.)

ADDING/ASSIGNING

With this bi-directional association properly modelled, if we have a Person called bob, we can add a Car called toyotaCorolla to bob's collection of cars in one of two ways:

```
bob.Cars.Add(toyotaCorolla);
```

or:

```
toyotaCorolla.Owner = bob;
```

These are, in fact, equivalent statements in Habanero, and both have the same effect on the object model in memory.

In the one-to-one Manager/Department example, if bill is the manager of a department called marketing then these two statements are equivalent (once again assuming the relationship is properly modelled with the reverse relationships configured):

```
bill.CurrentDepartment = marketing;
```

or:

```
marketing.CurrentManager = bill;
```

REMOVING/UNASSIGNING

In a similar way, if we have a Person called bob who has a Car called toyotaCorolla in his collection of Cars, we can remove it in one of two ways:

```
bob.Cars.Remove(toyotaCorolla);
```

or;

```
toyotaCorolla.Owner = null;
```

These again have the same affect as each other. You can think of them being the exact same operation, just two ways to express them.

If we wanted to replace bob's toyotaCorolla with a shiny new toyotaLexus we could do it in two equivalent ways:

```
bob.Cars.Remove(toyotaCorolla);  
bob.Cars.Add(toyotaLexus);
```

or:

```
toyotaCorolla.Owner = null;  
toyotaLexus.Owner = bob;
```

If bob sold his toyotaCorolla on to jim, we can indicate this as follows:

```
bob.Cars.Remove(toyotaCorolla);  
jim.Cars.Add(toyotaCorolla);
```

or:

```
toyotaCorolla.Owner = jim;
```

Finally, if bob sold his toyotaCorolla to jim and replaced it with a shiny new toyotaLexus:

```
bob.Cars.Remove(toyotaCorolla);  
jim.Cars.Add(toyotaCorolla);  
bob.Cars.Add(toyotaLexus);
```

or:

```
toyotaCorolla.Owner = jim;  
toyotaLexus.Owner = bob;
```

It is often simpler to use the single side of the relationship as it easier to read and understand, but it's up to you what to use as they all accomplish exactly the same thing.

In the one-to-one example of Manager/Department we can remove a manager (bill) from a department (marketing), and vice versa, as follows:

```
bill.CurrentDepartment = null;
```

or:

```
marketing.CurrentManager = null;
```

CREATING

For this we will use the example of an Invoice and Invoice Lines as it seems more intuitive to create an Invoice Line through an Invoice than create a Car through a Person.

If we have an invoice called invoice1000 and which to create a lines on it, we could write the following:

```
InvoiceLine line = invoice1000.InvoiceLines.CreateBusinessObject();
```

In Habanero it is also allowed to create an Invoice Line and add it to an Invoice, for convenience. To do this you could write:

```
InvoiceLine line = new InvoiceLine();  
invoice1000.InvoiceLines.Add(line);
```

Again, these two forms are equivalent. The Add method realises the Invoice Line is new and tracks it as a created object as if it were created through InvoiceLines.CreateBusinessObject().

In the case of a one-to-one relationship the first form is not applicable, and the second form becomes:

```
Person bill = new Person();
marketing.CurrentDepartment = bill;
```

MARKING FOR DELETION

Using the example of an Invoice and Invoice Lines again, if we wish to delete line1 of an invoice called invoice1000 we could write:

```
invoice1000.InvoiceLines.MarkForDelete(line1);
```

Or we could write:

```
line1.MarkForDelete();
```

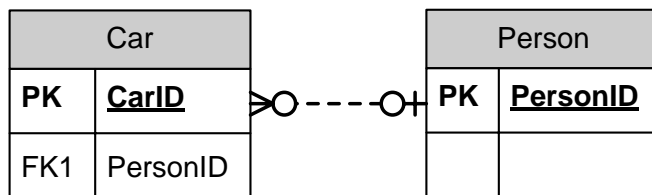
In the case of a one-to-one relationship the first form becomes:

```
marketing.CurrentManager.MarkForDelete();
```

The second form remains the same.

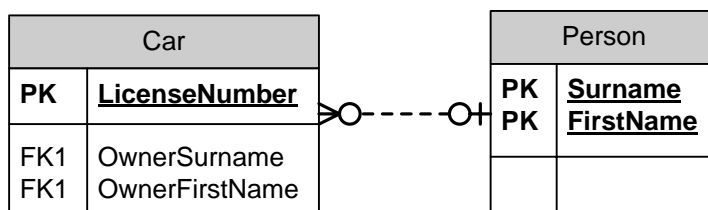
MAPPING TO A DATABASE MODEL

In order to make the relationship you have modelled persistent to a relational database, you need to also indicate what fields model the relationship at the database level. The above bi-directional relationship between Car and Person might be modelled in the database as follows:



Thus the Owner relationship between Car and Person would be a single relationship which relates the PersonID field on Car to the PersonID field on Person. Meanwhile the Cars relationship between Person and Car would be a multiple relationship which relates the PersonID field on Person to the PersonID field on Car. By indicating to Habanero what fields in the database model the relationship, it will ensure that the fields are always updated to the correct values when persons or cars are saved.

Habanero supports multi-field keys, so we could have the same domain layer relationship mapping to a set of tables that look like this:



This can be done by linking the relationships on multiple properties (OwnerSurname to Surname and OwnerFirstName to FirstName).

With these properties configured, when an object's relationship is changed in the domain layer, that change will be persisted to the database when the object or relationship is persisted. So, for example, consider an Invoice Line that is created in the following manner:

```
InvoiceLine line = invoice.InvoiceLines.CreateBusinessObject();
```

The created invoice line will be persisted when the Invoice is persisted, and the fields in the database that relate the InvoiceLine table to the Invoice table will be set correctly.

In the same way if a Car's owner is changed in code the relevant properties are immediately updated, ready to be persisted when the car (or the owner) is persisted.

MODELLING OPTIONS

Each relationship modelled in Habanero/FireStarter requires certain fields:

- Name: the name of the relationship. A property with this name will be generated on the class this relationship belongs to.
- Related class and related assembly: together these fields fully determine the related .NET entity type
- Type: this can be single or multiple, indicating the multiplicity of the relationship
- Related properties: these are pairs of property names that link a property of this class with a property of the related class, and are used in persistence and loading of the related objects.

A few more fields warrant some description too:

- Relationship type: this can be association, aggregation or composition. When modelling a bi-directional relationship only one of the directions should be modelled as an aggregation or composition – the reverse relationship should be left as association. The default relationship type is association.
- Reverse relationship: this is the name of the relationship that is the reverse relationship of this one (the other half of a bi-directional relationship). It is recommended to model this for all bi-directional relationships; Habanero is able to figure out the reverse relationship if you don't model it, but it can be mistaken if there is more than one relationship between two entities that are related on the same fields.
- Delete action: this option determines what happens to related objects when this relationship's owning object is deleted. The default option is Prevent. Note that not all options are available for all relationship types (see Behaviour). The possible options are:
 - Delete Related – delete all related objects as part of the same transaction
 - Dereference Related – clear the foreign keys of all related objects (set them to null) as part of the same transaction
 - Prevent – if any related objects exist disallow the delete process to continue
 - Do nothing – don't do anything in particular. This option is available to support circular relationships and to allow the option of using the underlying database's "cascade delete" options.
- Order by: an order by clause used to sort of the related objects as they are loaded. This only applies to multiple relationships
- Owning BO has foreign key: This only applies in a single to single (that is, a one to one) relationship, and is used to indicate which of these relationships holds the foreign key to the other (the default is true, so this should be set to false for the relationship that does not have the foreign key). Note that this field is very important in the case of a one to one as without it Habanero will throw an error when you try to set an object via the relationship.

BEHAVIOUR OF DIFFERENT RELATIONSHIP TYPES

Each of the different relationship types has a slightly different set of rules governing it reflecting the difference between the natures of each relationship.

ASSOCIATIONS

The following rules apply to associations in Habanero. These rules are described in the context of the above Car and Person example to make them more readable.

1. A Car can exist independently of a Person if they are related by an association relationship.
2. A new (unpersisted) or already persisted Car can be added to the Cars of a Person :
`person.Cars.Add(car)`
3. A Car can be removed from the Cars of a Person : `person.Cars.Remove(car)`
4. A Car can be created via the Cars of a Person : `person.Cars.CreateNewBusinessObject()`
5. A Car can be marked for delete via the Cars of a Person : `person.Cars.MarkForDelete(car)`.
6. A Person may be deleted even if it has Cars (unless the Prevent Delete setting is used). If a Person is deleted, the valid strategies for how this delete is applied to its Cars are: Prevent Delete, Dereference or Do Nothing. Cascading of deletes (Delete Related) is not allowed for associations.
7. A Person is considered to be dirty (that is, requiring persistence) if it has added, created, marked for delete or removed Cars
8. A Person is not considered to be dirty if it has any previously persisted Cars that are dirty
9. If a Person is persisted then its Cars relationship will be persisted, but not the Cars themselves. In other words, any created or new Cars will be saved and any marked for delete Cars will be deleted. Added or removed Cars that were previously persisted will only have their relationship properties persisted. If they have other dirty properties these will be left alone.

AGGREGATIONS

The following rules apply to aggregations in Habanero. These rules are described in the context of the Shipment/Package relationship described above.

1. A Package can exist independently of a Shipment if they are related by an aggregation relationship
2. A new (unpersisted) or already persisted Package can be added to the Packages of a Shipment :
`shipment.Packages.Add(package)`
3. A Package can be removed from the packages of a Shipment : `shipment.Packages.Remove(package)`
4. A Package can be created via the packages of a Shipment :
`shipment.Packages.CreateNewBusinessObject()`
5. A Package can be marked for delete via the packages of a Shipment :
`shipment.Packages.MarkForDelete(package)`.
6. A Shipment may not be deleted if it has Packages (although these may be removed by using an appropriate deletion strategy). The valid strategies for how this delete is applied to its Packages are: Prevent Delete, Dereference (remove), Delete Related, or do nothing.
7. A Shipment is considered dirty (that is, requiring persistence) if it has added, created, marked for delete or removed Packages
8. A Shipment is considered dirty if any of its Packages are dirty.
9. If a Shipment is persisted then all its Packages are persisted

COMPOSITIONS

The following rules apply to composition in Habanero. These rules are described in the context of the Invoice/Invoice Line relationship described above.

1. An Invoice Line cannot exist independently of an Invoice if they are related by a composition relationship
2. A new (unpersisted) Invoice Line can be added to an Invoice: `invoice.InvoiceLines.Add(invoiceLine)`. An already persisted Invoice Line cannot be added to an Invoice.
3. An Invoice Line cannot be removed from an Invoice
4. An Invoice Line can be created via the Invoice Lines of an Invoice: `invoice.InvoiceLines.CreateBusinessObject()`
5. An Invoice Line can be marked for delete via the Invoice Lines: `invoice.InvoiceLines.MarkForDelete(invoiceLine)`
6. An Invoice may not be deleted if it has Invoice Lines (although these may be deleted by using an appropriate deletion strategy). The valid strategies for how this delete is applied to its Invoice Lines are: Prevent Delete, Delete Related or Do Nothing.
7. An Invoice is considered to be dirty (that is, requiring persistence) if it has created (or new, added), or marked for delete Invoice Lines.
8. An Invoice is considered to be dirty if any of its Invoice Lines are dirty.
9. If an Invoice is persisted then all of its Invoice Lines are persisted.

SUMMARY

The table below compares the different relationship types side by side to highlight their similarities and differences. The table refers to parents and children in a general sense – the parent is the owner of a uni-directional relationships, while the child (or children) is (or are) the related object (or objects). For example, in the Person/Car relationship “a person can own zero or more cars”, the Person is the parent and the Cars are the children. In the Car/Person relationship “a car is owned by one person”, the Car is the parent and the Person is the child.

The essential difference between association and the other two relationship types is what is persisted when you persist the parent object – in an association only added, removed, created and deleted objects are persisted – all other children are not. In aggregation or composition, all children are persisted (if they are dirty/changed).

The essential difference between aggregation and composition is whether a child object can move from one parent to another. With aggregation the child object can exist without the parent or can be moved from one parent to another, whereas with composition the child is linked to its parent for good, never to be delinked.

	Association	Aggregation	Composition
Child can exist independently?	Yes	Yes	No
New child can be added?	Yes	Yes	Yes
Already persisted child can be added?	Yes	Yes	No
Child can be removed?	Yes	Yes	No
Child can be created?	Yes	Yes	Yes
Child can be marked for deletion?	Yes	Yes	Yes

Deletion strategies allowed	Prevent Delete; Dereference; Do Nothing	Prevent Delete; Dereference; Delete Related; Do Nothing	Prevent Delete; Delete Related; Do Nothing
Parent is dirty if has created children?	Yes	Yes	Yes
Parent is dirty if has added children?	Yes	Yes	N/A (new added children are considered created.)
Parent is dirty if has removed children?	Yes	Yes	N/A
Parent is dirty if has children marked for deletion?	Yes	Yes	Yes
Parent is considered dirty if its children are dirty?	No	Yes	Yes
Dirty relationships are persisted?	Yes	No (entire child is persisted, not just relationship)	No (entire child is persisted, not just relationship)
Dirty children are persisted?	No	Yes	Yes

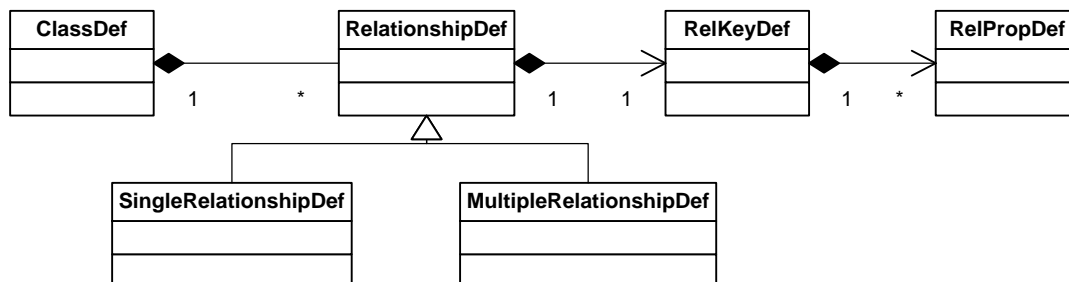
EXAMPLE

TODO

IMPLEMENTATION

Habanero's implementation of relationships uses a set of definition objects to model the definition of a relationship, and then another set of objects that represent the relationship itself at run-time – similar to the way its Business Objects are implemented.

RELATIONSHIP DEFINITIONS



When modelling relationships in FireStarter, or in XML, you are defining the structures of the relationship (that is the definitions). The definition of a relationship (the RelationshipDef) has a number of properties (such as the relationship type, the delete action, the related class and so on), and it contains a single RelKeyDef that

defines the key that the relationship will use to load the objects from a relational database. This in turn contains one or more RelPropDefs – one for each field in the key.

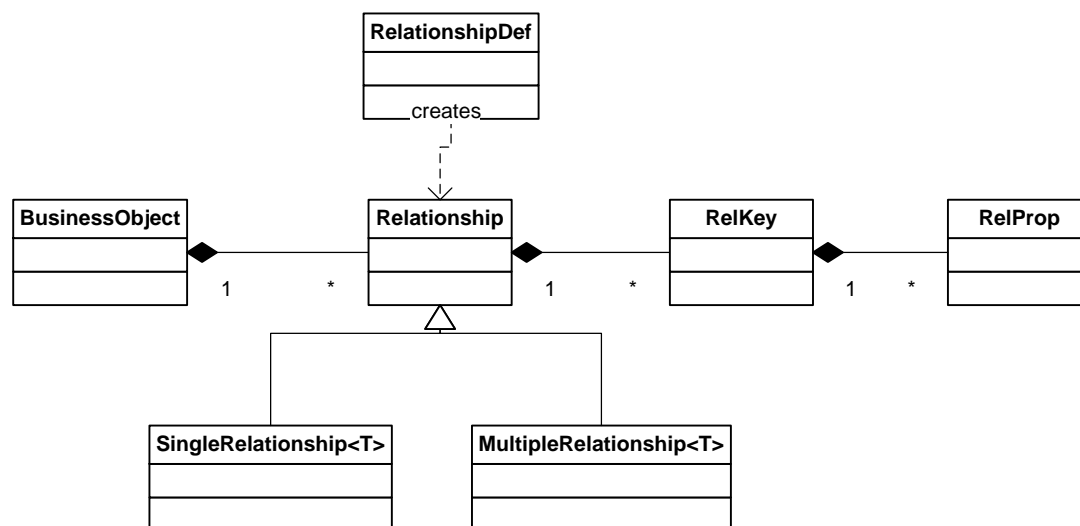
There are two subclasses of RelationshipDef: SingleRelationshipDef and MultipleRelationshipDef, each having the fields specific to that type of relationship.

A typical XML representation of the Car/Owner relationship would be:

```
<class name="Person">
  <property name="PersonID" type="Guid" />
  <primaryKey>
    <prop name="PersonID" />
  </primaryKey>
  <relationship name="Cars" type="multiple" relatedClass="Car"
    reverseRelationship="Owner" >
    <relatedProperty property="PersonID" relatedProperty="PersonID" />
  </relationship>
</class>
<class name="Car">
  <property name="CarID" type="Guid" />
  <primaryKey>
    <prop name="CarID" />
  </primaryKey>
  <relationship name="Owner" type="single" relatedClass="Person"
    reverseRelationship="Cars" >
    <relatedProperty property="PersonID" relatedProperty="PersonID" />
  </relationship>
</class>
```

Because the Car/Owner relationship is an association relationship, the relationship type is not required (associations are the default). Note that the complete model requires the reverseRelationship property to be set so that the relationships are linked together into one bi-directional relationship.

RELATIONSHIP INSTANCES



This diagram indicates what happens at runtime: when a BusinessObject is created it asks each RelationshipDef in its ClassDef to instantiate a Relationship of the correct type (either SingleRelationship<T> or MultipleRelationship<T> where T is the type of the related class), and this Relationship is added to its relationship collection. The RelKey and RelProp objects of a Relationship mirror the RelKeyDef and RelPropDef directly, and are simply instantiations of those definitions. These can be ignored for almost all purposes.

Each BusinessObject has a collection of Relationships which can be accessed by name via its Relationships property. The SingleRelationship's SetRelatedObject and GetRelatedObject can be used to set or get the object on the other end of the relationship, while MultipleRelationship's BusinessObjectCollection retrieves the collection of objects at the other end of that relationship.

Most use of relationships when writing client code is done by using the properties generated by FireStarter, and as such the above structures should remain invisible to you. All the loading, updating of properties including the setting of foreign keys, mapping to the database and so on, is done behind the scenes – you need only concentrate on the domain model itself.

ACCESSING RELATIONSHIPS DYNAMICALLY

It can happen that you need to configure a relationship to work in a specific way at run time, perhaps depending on a setting set by the user or some business rule. It is always possible to access the collection of relationships of a Business Object using the Relationships property.

As an example, let's say I want to write a general purpose utility to dump an entire object tree (of unknown type) to a text file, for debugging or troubleshooting purposes. I also only want to dump related objects if they are related by composition, and ignore aggregations and associations.

```
public void Dump(IBusinessObject bo, Stream stream)
{
    StreamWriter writer = new StreamWriter(stream);
    writer.WriteLine("----- {0} : {1}", bo.ClassDef.ClassName, bo);
    foreach (IBOProp prop in bo.Props)
    {
        writer.WriteLine(prop.PropertyName + ": " + prop.Value);
    }
    foreach (IRelationship relationship in bo.Relationships)
    {
        if (relationship.RelationshipDef.RelationshipType !=
RelationshipType.Composition) continue;
        writer.WriteLine("--- Relationship : {0}",
relationship.RelationshipName);
        IMultipleRelationship multipleRelationship = relationship as
IMultipleRelationship;
        if (multipleRelationship != null)
        {
            foreach (IBusinessObject relatedBO in
multipleRelationship.BusinessObjectCollection)
            {
                Dump(relatedBO, stream);
            }
        }
        ISingleRelationship singleRelationship = relationship as
ISingleRelationship;
        if (singleRelationship != null)
        {
            IBusinessObject relatedObject =
singleRelationship.GetRelatedObject();
            if (relatedObject != null) Dump(relatedObject, stream);
        }
    }
}
```

```
}
```

In this sample code, first the type of the Business object is dumped, then the properties, and then the related objects. There is a little work done to figure out whether each `IRelationship` is an `ISingleRelationship` or an `IMultipleRelationship`, and to do the appropriate operation for each, and there is one check to see if the relationship is a composition one, but overall the code is really simple.

Of course, you can develop as if all these constructs are not there (which is a good thing – you need to be concentrating on your domain model rather than the infrastructure), but when you need to do something generic like this it's entirely possible to write code that applies to each and every object without any complex reflection logic, a very powerful possibility.

SINGLE RELATIONSHIP BEHAVIOUR

We've already seen how the single relationship is used in client code:

```
Person owner = car.Owner;
```

The `Owner` property itself has been generated as follows:

```
public virtual Person Owner {  
    get {  
        return Relationships.GetRelatedObject<Person>("Owner");  
    }  
    set {  
        Relationships.SetRelatedObject("Owner", value);  
    }  
}
```

The `Relationships` collection is passing the `GetRelatedObject<Person>(string relationshipName)` call through to the `SingleRelationship<Person>` called `Owner`, and the same for the `SetRelatedObject(string relationshipName, object value)` call.

GETRELATEDOBJECT

The first time the `GetRelatedObject<T>()` (or `GetRelatedObject()`) method is hit, the relationship loads the related object from the data source. From then on, the loaded object is cached and remains cached in memory as long as its owning Business Object is cached. In the example above, once the `Owner` property's `get` is called the owner will stay in memory as the car stays in memory. Further calls to `GetRelatedObject` result in the cached object being returned, unless something is changed.

If by some chance the properties that are relating the two objects to each other are changed manually, for example:

```
car.OwnerID = newOwner.OwnerID;
```

or, if the `OwnerID` is not settable (as is correct):

```
car.SetPropertyValue("OwnerID", newOwner.OwnerID);
```

This code has bypassed the relationship which is not recommended, but could be useful if performance is vital such as for an import procedure (although this should only be decided after actual performance analysis). When `GetRelatedObject` is later called a quick check is made to see if the "foreign key" properties have changed, and if they have then the object will be reloaded using the new properties.

SETRELATEDOBJECT

This method is used to change the current related object. Calling it triggers off a `GetRelatedObject()` call so that the relationship knows if the object being set is really a new object or is the same as the current one. If the object being set is the same, nothing happens, but if it's new then the previous object will be removed from any reverse relationship and the new one added to any reverse relationship. We saw this earlier in the example of bob and jim. If bob is the current owner of `toyotaCorolla`, and we do the following:

```
toyotaCorolla.Owner = jim;
```

`SetRelatedObject` will remove `toyotaCorolla` from bob's cars (this is the reverse relationship of the `Owner` relationship) and add it to jim's cars. This is why modelling the reverse relationship is of vital performance to a sensibly behaving domain model.

MULTIPLE RELATIONSHIP BEHAVIOUR

We've already seen how the multiple relationship is used in client code:

```
BusinessObjectCollection<Car> cars = person.Cars;
```

The `Cars` property on the `Person` class has been generated as follows:

```
public virtual BusinessObjectCollection<Car> Cars
{
    get {
        return Relationships.GetRelatedCollection<Car>("Cars");
    }
}
```

The `GetRelatedCollection<Car>` call on the `Relationships` collection finds the appropriate collection called `Cars` (which is a `MultipleRelationship<Car>`), and returns the `BusinessObjectCollection` property, thus returning a `BusinessObjectCollection<Car>`.

BUSINESSOBJECTCOLLECTION

Calling this property, as the above example code does, causes the collection of objects that make up this relationship to refresh (or load if it is the first time). This happens every time the property is accessed – the database (or other datastore) is hit each time to keep the relationship collection thoroughly up to date. This obviously has an impact on performance, so where necessary a reference to the returned collection should be retained.

It is possible to get at the collection without refreshing it by calling `CurrentBusinessObjectCollection` (accessible via the `IMultipleRelationship` interface). This property returns the current collection as is, so if you have not ever loaded it (via a call to `BusinessObjectCollection`) it will be empty.

It is worth noting that although the return type is `BusinessObjectCollection`, the actual object returned is a subtype of that, the `RelatedBusinessObjectCollection`. This is important because the `RelatedBusinessObjectCollection` overrides some of the behaviour of the `BusinessObjectCollection` when it comes to adding, creating, removing and marking objects for deletion. The rules governing this have already been discussed (see Behaviour of different relationship types), but suffice to say that these collections do not always act exactly the same as a normal `BusinessObjectCollection` as discussed in Chapter 4, and in fact act differently depending on what relationship type the relationship is.

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 Customer's 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 avoided 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:	<ul style="list-style-type: none"> It is time to start work on an order.
Normal Course	
1.	Open the order management section of the system
2.	The system will display all orders
Add a order to the system	
3.	Select Add
4.	Open a blank order form with the following details: <ul style="list-style-type: none"> Order # - Unique number generated by the system Order Description Date Initiated - The date that the matter was accepted by the Order department
5.	

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 usefulness 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.

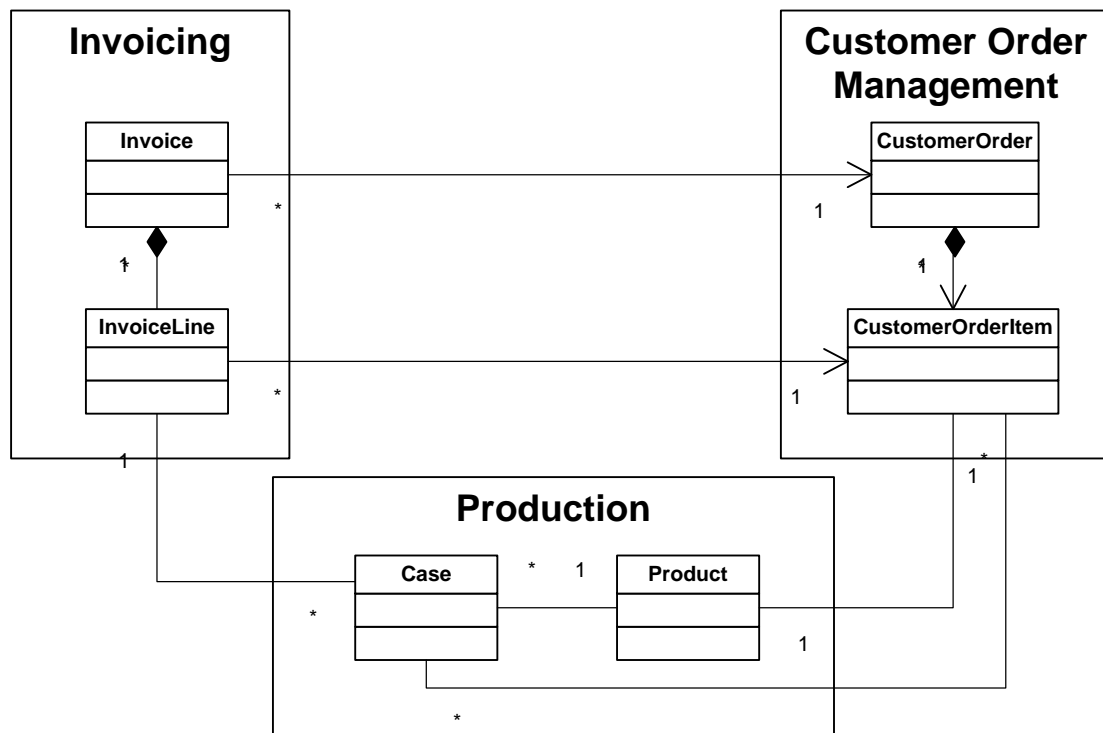
What gets modelled in this? The business objects of interest to the business, the relationships to them, the rules for updating editing and deleting business objects as well as security relating to using the business objects.

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.

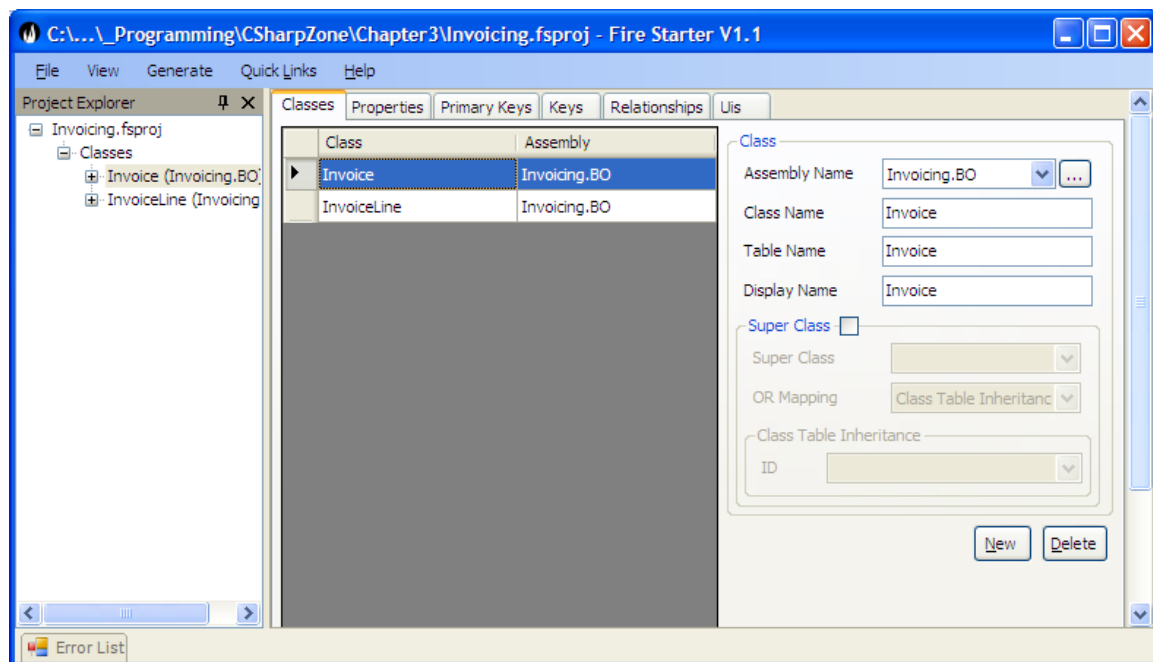
There are many more rules for this system but these have been selected as examples of rules that are modelled in the domain layer. In addition the system has been separated into separate sub systems as per this diagram. For the purposes of this example the Customer and all related rules and data have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

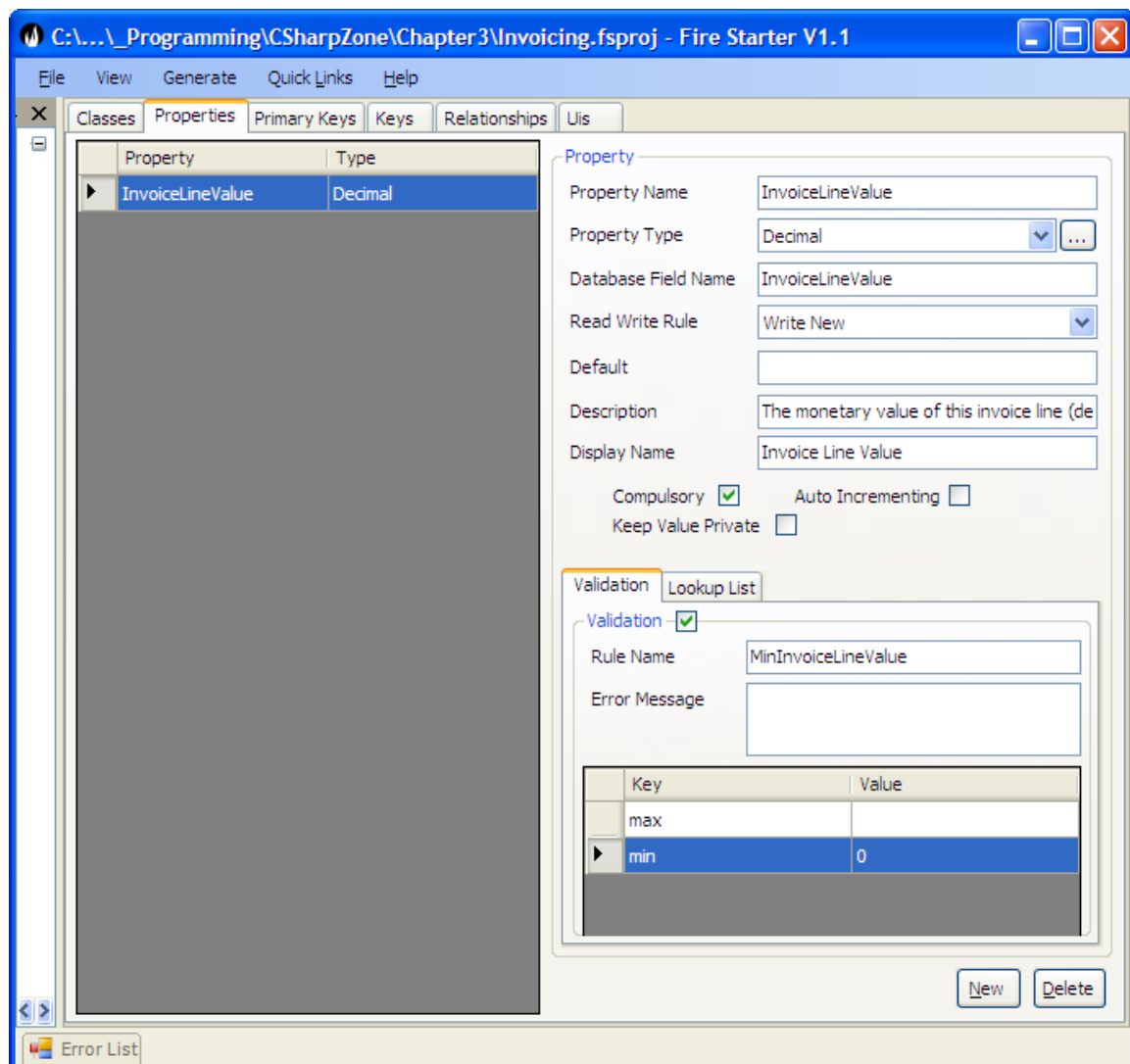


Then we model the various properties that each Business Object has in our case the Invoice Line has a line number – the sequence number of the line printed on the invoice, a line description – a textual description of the invoice line and a value the amount that the customer is being charged.

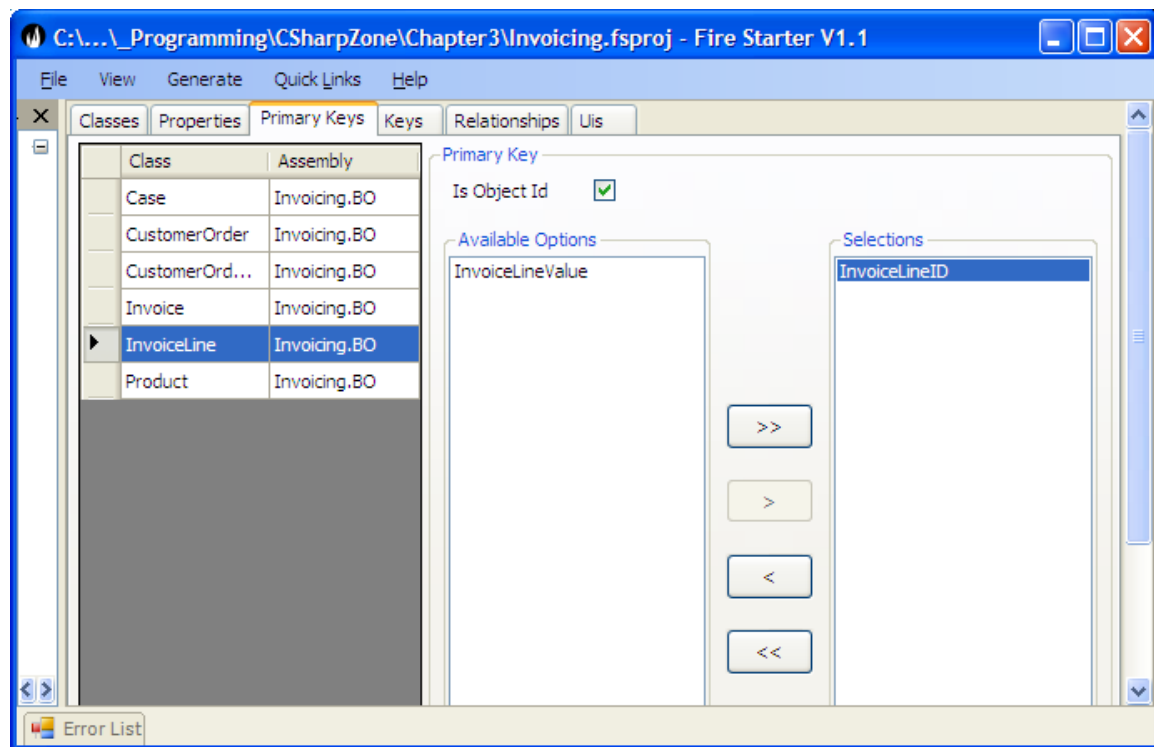
Each of these properties of a business object can have rules set for it. We will demonstrate this by adding the property *Invoice Line Value* to *Invoice Line*.

From this we can see that the *InvoiceLineValue* is

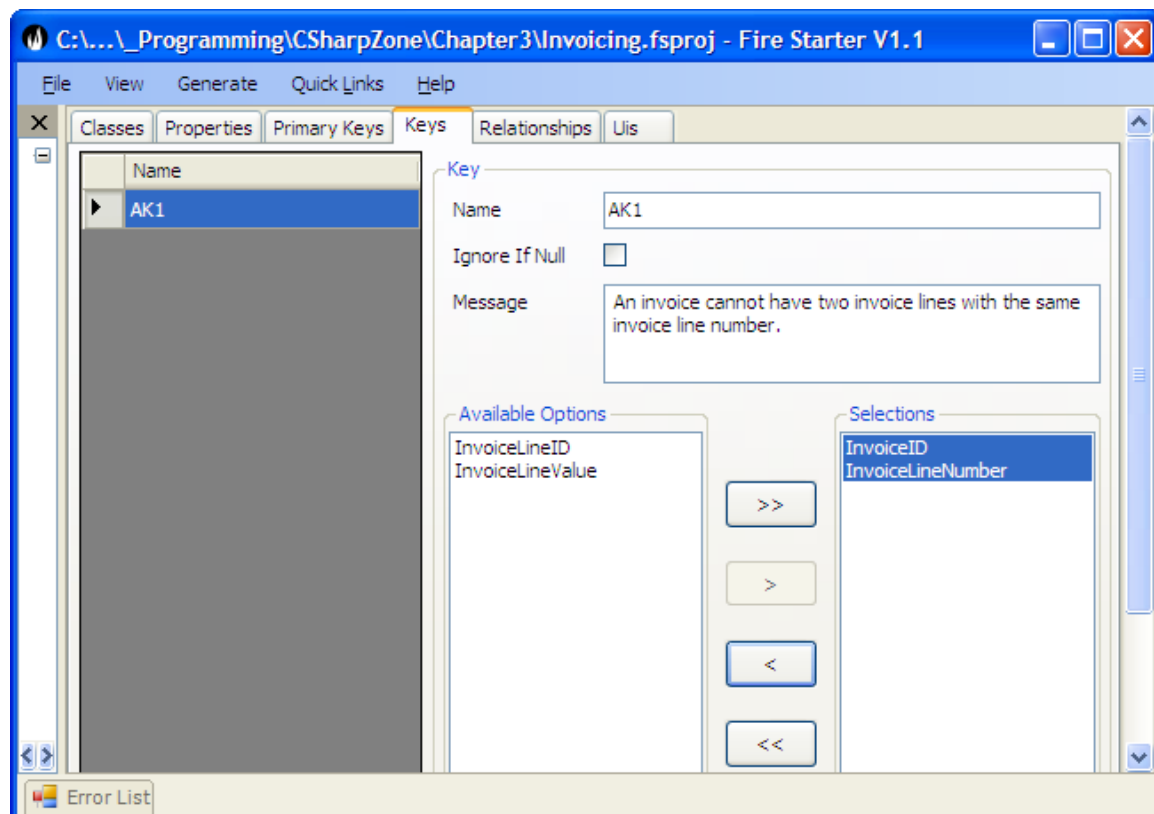
- Type Decimal.
- Read Write Rule of Write New i.e. it must be set to a valid value before the object can be persisted for the first time else the object will be in an invalid state. The *InvoiceLineValue* can never be updated after the object has been persisted to the database. The other options for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



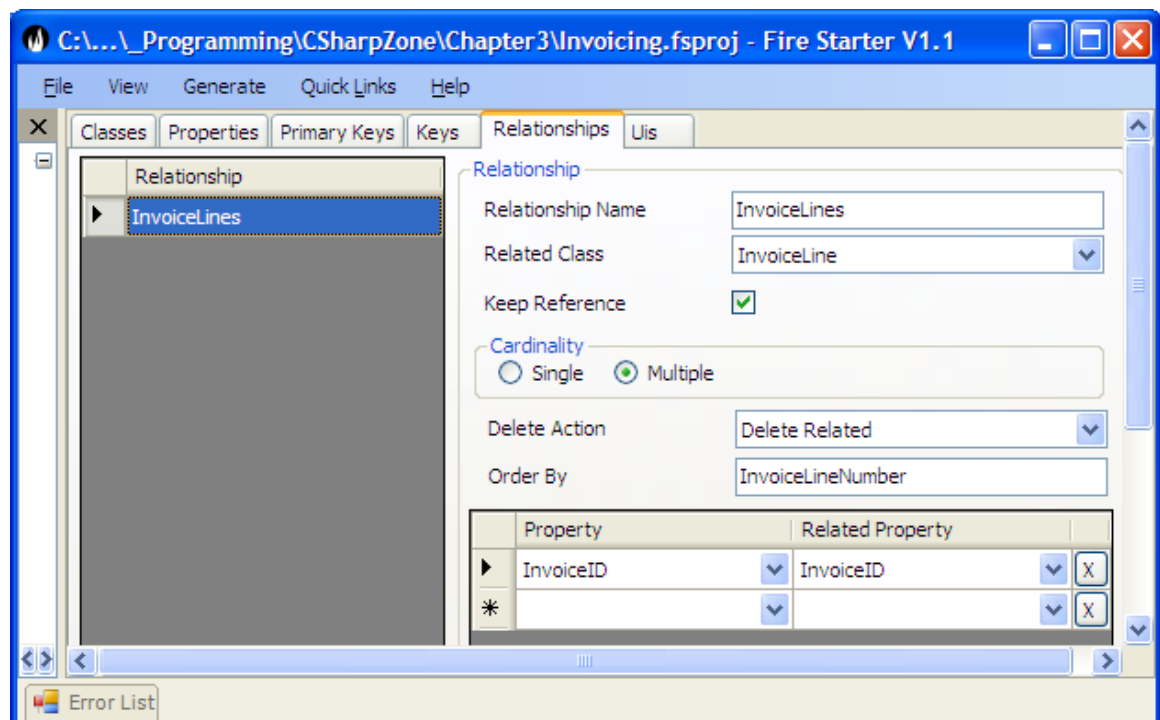
For each object you can also identify alternate keys. This is a single or a group of properties that are unique for the Business Object and act as a constraint to ensure that two Business Objects cannot be persisted with the same values for the alternate key. E.g. An invoice cannot have two invoice lines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