Marina Manager

a software solution

Ben Cotton

8/11/2004

Acknowledgments

The author wishes to acknowledge the software development opportunity offered by Chris Hawkes, supervisor for the Nelson Marina, which provided the basis for this report. His detailed documentation of requirements, generosity with his time, and praise were also very helpful.

The invaluable guidance of the supervising tutor for this project, Matthias Otto, is also acknowledged.

Lastly, the author wishes to thank his father, Roger Cotton, for proofreading and offering helpful suggestions.

Terms of Reference

As part of the Bachelor of Information Technology programme at NMIT students are required to undertake a project (PRJ300) of 450 hours of work, in a area of their own choosing. The aims of this project, as outlined in the project guidelines document, are:

- to provide students with the time to undertake a significant piece of work in an area of interest to them
- to provide students with an environment in which they can develop their problem solving skills to a high level
- to provide students with an opportunity to develop their expertise in one or more specialised areas of information technology, and
- to give students experience in communicating their work to others in both written and oral forms.

The report produced as part of the project is intended to be a final, formal summation of the work carried out, and is to be written with the information technology literate reader in mind.

Table of Contents

Table of Figures

1. Executive Summary

While information technology has revolutionised many areas, for some reason or another many are still untouched by it. The question is why?

This report covers the development of an information system to manage a small marina, which is currently administered through the use of paper records and whiteboards. Due to ongoing expansion of the marina, and the employment of an assistant to the Marina Supervisor, the solution needed to allow for growth, and include the ability to access data from multiple locations.

The author had several personal goals in mind when carrying out all phases of the project. These were:

- to gain experience with a relatively unfamiliar development environment
- to make the solution cross-platform in design, and
- to make parts of the system reusable in future projects.

Regular meetings were scheduled with the client (the Marina Supervisor) throughout the project work. Use cases and prototypes were discussed in these meetings. The client also provided a wish-list of requirements.

Behind the scenes, research was carried out in relevant subject areas, and database models and class diagrams were generated.

From preliminary research into existing marina management systems, there was found to be a sharp divide between low end systems—which are largely aimed at people inexperienced with computers—and high end systems.

A variety of applicable web application technologies were also researched.

After preliminary research the remaining project work was divided into three phases: database analysis and design, software design, and implementation.

Analysis showed the marina structure to be quite complex in nature. It is made up of a variety of different berth types, both on the water and off, which can be allocated for different types of lease. Types of lease can include subleases (when

long term leasers are temporarily away) and joint leases (i.e. one extra large vessel occupying two adjoining berths). Each type of lease has a different fee structure associated with it.

As mentioned earlier, the marina is in the process of expansion—new berths are currently being added.

The system produced needed to be able to:

- store information about the vessel and vessel owner in each berth
- provide some kind of graphical display of the marina berths and their statuses
- project the status of berths, to see when they would be available
- handle payments, including credits
- print missives such as receipts and invoice notes; and
- print reports such as graphs and listings of vessels which meet certain criteria.

The major difficulty that arose during database design was finding a good way of handling subleases and joint berth leases. Ultimately, the design was made to revolve around a central *Application* table which "contains" leases, temporary absences and payments, and through which subleases are linked to their parent leases.

In terms of the interface design, this led to the creation of a central, multi-tabbed *Application* form for editing all aspects of a given application, including its leases, absences and payments.

Other features of the interface design included:

- forms for displaying lists of applications, leases and a graphical representation of the berth statuses for any given date.
- forms for setting up berths and fees
- a form for printing reports, and
- a graphical code used throughout to represent the different types of applications, berths and leases.

In terms of the software design, the system was split into three layers or packages: a data access layer, a business logic layer, and a user interface layer. In addition, an external package was included containing generic functionality which could be used in future projects.

When it came to actually implementing the system, a number of possibilities for the target platform were considered. Eventually, it was decided that the resultant system would be a desktop application, written in C# on Microsoft's .NET Framework. Microsoft SQL Server Desktop Engine was chosen to be the database management system.

Once the platform had been selected, numerous adaptations were made to fit the design to the platform, and suitable debugging techniques were found.

No formal testing was performed, but basic tests were carried out during implementation and the client was urged to notify the author of any issues found in prototypes.

On completing the final prototype the author was very satisfied with the capabilities and quality of the final prototype, and this was reflected by the client's praise. However, the author also found that:

- communications with the client could have been better (these were largely due to the lack of computer literacy on the part of the client)
- the sequence of the analysis and design phases was not optimum
- too much emphasis was placed on the cross-platform aspect of the design work
- the class designs were very cohesive, manageable and reusable
- the target platform was excellent, and
- the development environment used was buggy.

In conclusion, the project work served to demonstrate three key facts about the information technology industry: the value of object-oriented design, the importance of good scope management, and the necessity of training. A lack of acceptance of these facts could explain why information technology hasn't yet penetrated all areas.

Although the project was mostly successful, more could have been achieved had there been a better balance between the author's personal goals (especially the goal of a cross-platform design) and the client's requirements.

2. Introduction

Information technology, as the term implies, is revolutionising the management of information across all industries. Tasks which once required hours of work and screeds of paper can now be accomplished in seconds, without any paper at all.

While the information technology (IT) revolution is advancing at a tremendous pace, its penetration into all areas is far from complete. Many tasks which could be done far more quickly, efficiently and reliably using information technology are still being done with the old standards.

Why are these gains still hypothetical for a great many endeavours? Perhaps the people involved are unaware of the rewards that can be extracted from information technology. It could also be precisely this unfamiliarity which alienates them.

Along with pens, pencils and paperwork, there is another common element that is also being (inadvertently) marginalised with the IT revolution: human interaction. Throughout history, humans have relied on human-to-human networking to get jobs done. In contrast, the largest network on the planet nowadays consists, in practical terms, entirely of machines.

Considering business and other affairs have always involved direct human interaction, it is only natural for the adoption of technology to be slow. After all, computers are ultimately glorified calculators, with no real intelligence of their own—their power lies in the precision and speed with which they can tackle problems.

At the end of the day, information technology is about the sculpting of hardware and software to manage human affairs. One such application of technology—the management of a marina—forms the basis of this report.

3. Problem Description

The Nelson Marina is a 24 hour, seven-day-a-week operation which is owned by the city council and managed by a contracted supervisor.

Currently the allocation of berths is managed through the use of a paper-based filing system and whiteboards which show representations of berths and their statuses. This has been found to be error-prone, inefficient and inflexible. The goal of this project was to create an information system to replace the current system.

As the marina is expanding (this expansion includes the employment of an assistant to the Marina Supervisor), room for growth and the ability to access to the data from multiple locations needed to be included in the design.

4. Personal Goals

Prior to the commencement of this project, and indeed, prior to his current studies, the author had garnered a reasonable amount of non-professional experience in information technology, particularly programming. While unaware of the methods of object-oriented software *design*, the author was sufficiently familiar with object-oriented development environments and languages to form strong opinions and curiosities—especially about emerging platforms.

4.1. Development Environment

As a result of these strong curiosities about development environments, it became a personal goal to try something new, even if it was just something which hadn't been covered in the papers taken.

The author's main experience lay in the Borland Delphi and C++ programming with Microsoft Visual Studio. Neither of these had been used in the course of his studies, in favour of Microsoft Visual Basic 6 and .NET. Because of this, and the strong negative opinion formed about Visual Basic—particularly Visual Basic 6—it was decided that development should at least be performed on one of the first two platforms, or else on an unfamiliar platform.

4.2. Cross-platform Design

The author has also always held an admiration for cross-platform design, if not the resolve to follow through with those ideals. In the interest of not only pushing his own skills to the limit, but of producing something which wasn't tied down to just one platform, it was decided that the product of the project work would be as cross-platform in design as possible.

4.3. Reusability

Finally, following the aspirations of object-oriented design, it was the desire of the author that any widely applicable functionality developed for the prototype would be made accessible and reusable for future projects.

5. Methodology

The basic execution plan for project work, as mapped out before the project was even started, consisted of four phases: preliminary research, database analysis and design, software design, and software development (i.e. implementation). As originally envisioned there was no overlap between the phases, but it was expected that work carried out in the earlier stages would inevitably need to be revised in the later ones. The plan was never intended to be followed dogmatically; it was merely a natural framework for the work that then lay ahead.

5.1. Client Relations

From the beginning, meetings were conducted with the client (the Marina Supervisor) on a regular basis. These meetings (seven in total) enabled the author to discuss the requirements, ask questions, float ideas and demonstrate and explain the prototypes produced.

5.1.1. Requirements

As mentioned, a major topic of discussion during the meetings with the Marina Supervisor was the requirements for the system. In this regard the author was very lucky, as the client produced a document describing the marina, as well as a wish-list of features. Documentation supplied by the supervisor also included sample forms and reports.

5.1.2. Use Cases

Use cases were used to structure the given requirements into a more easily usable form, and bounce them back at the client, for the purposes of verification. They also acted as a springboard into the discussion of common scenarios that which the resultant system would need to handle.

The use cases for this project can be found in Appendix E.

5.1.3. Prototypes

From the requirements and use cases, a range of prototypes were created.

In the software design phase of the project, a dumb graphical user interface was created. This was demonstrated to the client to train them up for the eventual product, and also to allow them to suggest improvements and clarify aspects of the requirements.

Several working prototypes—incremental builds created during the actual implementation phase—were also demonstrated to the client. With these prototypes much greater emphasis was placed on training and quality control. The client was able to describe various scenarios and their correspondence with the features of the product was explained by the author. The author also asked that the client retain and play around with the prototypes to reveal any issues or bugs.

5.2. Development

Behind the scenes several tools and techniques were used in the course of the development. A description of the most important ones follows.

5.2.1. Research

Before any real analysis or design work was actually undertaken some research was carried out on existing systems used in the field of marina management, as well as technologies applicable. The findings of this preliminary research gave a helpful insight on the problem at hand.

Limited research was also done during the other phases of the project to find relevant tools, techniques and strategies.

5.2.2. Database Models

For this project three database models were developed: a conceptual model, a logical model and a physical model. Each of these models contains more implementation specific details than the previous, so that the database design can be easily comprehended and adapted to different platforms—thus going towards the author's personal goal of a cross-platform design.

Though the types of database model lend themselves to sequential development, as was the case in this project, to a significant degree the model development was concurrent. It was carried out this way as each model is, in effect, a different perspective on the same design, and thus brought to light different issues that affected all of the models.

5.2.3. Class Diagrams

Given that a large portion of the project work revolved around software design and implementation, UML (unified modelling language) class diagrams were naturally essential, at least for the design phase. Preliminary class diagrams were created during the design phase and these modelled all parts of the software design, except classes for the actual forms.

The class diagrams of the final prototype (located in Appendix F) were created using the reverse engineering functionality of the development environment used, which created the necessary UML from the program code. The reason for this is that it would have been too time-consuming to keep class diagrams up-to-date throughout implementation. The final class diagrams were included as documentation for future reference.

6. Preliminary Research

As mentioned previously, preliminary research into software solutions for the management of marinas was undertaken. Three broad areas were considered: shrink-wrapped products, systems currently in use at marinas, and technologies that could possibly apply to the problem.

6.1. Shrink-wrapped Products

In the process of searching the Internet, the author found a number of existing marina management systems on the market. In fact, it quickly became apparent that the market is saturated with such solutions. From the products the author was actually able to investigate in the time given, one major observation could be made: the market is clearly divided into a low end and a high end.

An example of an archetypical high end system would be Execu/Tech's HOTEL[1], which comes in Professional and Premium versions. As the name suggests, it was "designed for and by high profile 5 star" hotels, but along with hotel rooms, it can also apparently manage marina berths. The description mentions advanced features such as integration with point-of-sale products, multi-platform networking ability, inventory and accounting, including credit card processing.

[1] *Marina Management: marina software, marina management software.* Retrieved 7 July, 2004 from http://www.execu-tech.com/marinas.shtml.

In contrast there are products such as Marina Manager[2] (no relation to the project prototype), which merely provides the functionality for maintaining records about vessels and their owners; it doesn't manage the allocation of berths. Rather oddly, it features an integrated word processor, calculator and calendar tools—obviously intended to cater for small operations run by people who are largely inexperienced with computers.

6.2. Systems in Use

Considering the problem of marina management around which the author's project revolves, it is not all that surprising that there is a market for systems such as that being developed here. Further evidence of the relatively low tech world of small marinas was found in the sole response to a query about systems used that was sent via email to several marinas around the country.

Tauranga Bridge Marina[3] uses a Microsoft Access database to maintain records on the berths and the vessel in each berth, NZAGold for accounting and Microsoft Excel to record long term stays. However, the actual allocation of berths is done manually, and the their "graphical display system" for berths is a whiteboard with magnetic tabs.

6.3. Technologies

The nature of the problem, at first glance, suggested some kind of solution involving the Internet to make the data accessible from different locations. To that end, various web applications platforms were investigated including Java[4], ASP.NET[5], PHP[6], Perl[7],

---

[2] *Marina Management Software – Marina Manager.* Retrieved 10 July, 2004 from http://www.marina-management-software.com.

[3] Julie Bailey, Marina Administrator, Tauranga Bridge Marina. Personal correspondence.

[4] *Java Technology.* Retrieved 14 July, 2004 from http://java.sun.com.

[5] *ASP.NET Web: The Official Microsoft ASP.NET Web Site.* Retrieved 17 July, 2004 from http://www.asp.net.

[6] *PHP: Hypertext Preprocessor.* Retrieved 16 July, 2004 from http://www.php.net.

Python[8]. Development environments were also looked at briefly.

The first significant trend found was that almost all of the technologies feature the integration of code with HTML. The major exception to this is ASP.NET when development is carried using Visual Studio—the free Microsoft ASP.NET Web Matrix and other development environments for ASP.NET don't allow for the separation of code and HTML.

PHP, Perl and Python form part of the common open source LAMP platform (Linux, Apache, MySQL, PHP/Perl/Python), consisting of open source products, and as such, are cheap and ubiquitous. Other more obscure web scripting technologies found—TCL, Pike, Ruby and Lisp—are also quite low cost, open and cross-platform[9].

In contrast, ASP.NET costs a fair amount, and although theoretically cross-platform, it hasn't been ported yet (open source ports such as Mono[10] are in development); Java Server Pages requires it's own cross-platform server; and the more obscure ColdFusion scripting technology[11] has significant licensing fees.

---

[7] *The Perl Directory – perl.org.* Retrieved 16 July, 2004 from http://www.perl.org.

[8] *Python Programming Language.* Retrieved 17 July, 2004 from http://www.python.org.

[9] VerBeek, T. (2003). *Visual Basic, Active Server Pages, and other web scripting technology.* Retrieved 15 July, 2004 from http://microsoft.toddverbeck.com/script.html.

[10] *What is Mono?* Retrieved 16 July, 2004 from http://www.mono-project.com.

[11] *Macromedia ColdFusion MX.* Retrieved 15 July, 2004 from http://www.macromedia.com/software/coldfusion/.

7. Analysis

Through discussions with the client and readings of the requirements documentation, a rather complex picture of the marina operations and the desired functionality emerged. It should be noted that a lot of the seeming arcane-ness is probably attributable to an intricate relationship between the Marina Supervisor (the client), who operates the marina, and the city council, which actually owns the marina.

7.1. Marina Structure

At the time of writing, the Nelson Marina consisted of around 500 berths of various types (on the water and on land). Each of these berths can be leased, and there are several different leasing structures.

7.1.1. Berths

There are basically six different types of berth: permanent, visitor, pile mooring, hardstand, storage and swing mooring.

The terms permanent and visitor here refer specifically to pontoon berths; each pontoon berth may be allocated for either long-term or short-term stays, with different lease structures. All the other types of berth are not reserved exclusively for any particular term of stay.

Each berth has an identifying number associated with it. The berth numbers consist of a two letter code for the type of berth, or, in the case of pontoon berths, the letter of the pontoon; and two digits for the actual number of the berth.

Berths may have a length associated with them, but a given berth’s length doesn’t necessarily limit it to vessels of the same length or smaller, as larger vessels can sometimes be squeezed in.

Almost all of the pontoons are reasonably symmetrical, with the berths down the left hand side being oddly numbered and the berths down the right hand side evenly numbered (this is from the perspective of an observer on the shore facing the seaward end of a given pontoon). Unfortunately, an abnormally formed “I” pontoon breaks these rules: it has a bent shape, with only a few odd numbered berths (one of which is on the wrong side), and a range of alignments for the berths.

Extra pontoons are being added, and preparations are being made for extra dry land compounds for hardstand and storage berths.

Finally, swing moorings are handled fairly differently in comparison to the other types of berths—only a register of their current owner, location (GPS), and safety inspection status is maintained.

7.1.2. Leases

As alluded to earlier, the permanent and visitor allocated pontoon berths can be occupied by long-term and short-term leasers, respectively. Complicating matters is the fact that there is a third kind of lease—a temporary lease—and permanent berths can be subleased.

Temporary leases are basically interim leases granted when there are no permanent berths available, and the leaser is looking to stay long-term. One reason for the distinction is that the fee structure is different; another is the fact that temporary leases can be granted on berths allocated for visitors or already-leased permanent berths where the leaser is temporarily absent.

Already-leased permanent berths where the leaser is temporarily absent can be subleased to visitors or temporaries. When subleased, any fees paid by the sub leaser get credited to the permanent leaser’s account.

As a final complication with regards to the pontoons, the adjacent pairs of berths on the ends of almost all of the pontoons can be leased separately (as is usually the case), or as one extra long berth for unusually large vessels.

Non-pontoon berths, barring swing moorings, have associated leases similar to the pontoon permanent (i.e. long-term) lease. The only real differences are in terms of fees.

7.1.3. Fees

The different types of leases have different fee structures; however, the final prototype was only required to calculate daily fees. Visitor, pile mooring and hardstand leases are charged at a daily rate. In the case of visitor leases, the actual fee charged is dependent on the length of the vessel and the length of the berth.

7.1.4. Applications

With certain lease types, the leaser must formally apply for the lease; this entails some added fees in the form of straight application fees, as well as development levies. In the case of development levies, the leaser only has to pay once ever for each berth they lease.

A register is also kept of vessel owners who are “waiting” for berths—waiting in a rather loose sense—as some of entries date back to the 1970s.

7.2. Functionality Requirements

Thankfully, the client was quite flexible with regard to requirements—their expectations were relatively low. The only essential feature was the ability to access the data from different locations—some misinterpretation occurred here concerning the exact nature of this accessibility, which is discussed further in 9.1.

As mentioned in 5.1.1, a wish-list of features was created; these are summarised below.

7.2.1. Recorded Information

It was the stated desire of the client that the system be able to record a vast array of information about each berth's occupying vessel and vessel owner. This included (but was far from limited to) information such as the vessel's name, dimensions, and contact details.

7.2.2. Graphical Display

Some kind of graphical display of the pontoon berths and their current occupation status was wanted, though not mandated. The client would at a minimum be able to see the name of the vessel leasing any given berth, the type of lease, and whether the berth was actually available for subleasing.

7.2.3. Projection

The system would be able to project into the future and see which berths were available and when. This would allow the client to book berths for customers in advance.

A history would also be maintained for each vessel and each berth, containing information about the berths occupied and the vessels occupying, respectively.

It should be noted that the idea of having a static "view" of the marina berths for the current date, espoused by 7.2.1 and 7.2.2, was thought by the author to be unsuited to the task, given these projection requirements. The reasoning behind this was that any "view" of the marina berths could be efficiently dynamically generated from lease start and end dates stored in the system.

7.2.4. Payments

As mentioned in 7.1.3, fees owing would automatically calculated for certain lease types, on departure. In the case of subleases, the regular occupier of the relevant berth would be automatically credited the same amount.

All payments made (including credits for future leases) would be recorded, and the client would be able to edit them.

7.2.5. Missives

Missives such as receipts, invoice notes for the council and cancellation notices for permanent leases would be able to be printed out. In the case of invoice notes, the client would be able to control which charges were listed.

7.2.6. Reports

The system would be able to print statistical reports on annual (in terms of the financial year) and quarterly numbers of leases and berths.

According to the client's own wish-list, the ideal system would be able to print out listings of vessels without electrical warrants of fitness. However, talks suggested that a wide range of listings of subsets of records within the system may eventually need to be producible.

7.2.7. Graphs

Finally, the ideal system would be able to print out graphs of the numbers of visitors against the months of the year, and the numbers of permanent leases against months of the year.

Discussions with the client brought up a number of common scenarios which the ideal system would have been expected to be able to handle. Mostly these just further underlined the aforementioned requirements, however one scenario brought to light a significant issue.

The scenario in question revolved around the handling of incoming vessels late at night, which normally radio the Marina Supervisor for a berth. Very few details about the vessel and their owner are collected over the radio, let alone lease and application details. Because of this, the system developed needed to be able to deal with very incomplete data.

8. Design

Right from the start it was clear that the intricate relationships between berths, leases, vessels and other entities required a very inter-connected database, interface and business logic. The inter-connectedness is most evident in the database designs.

8.1. Database Design

The entities, attributes and relationships in the various database designs were roughly transcribed in the usual fashion from the business rules found in analysis. Explanations now follow for the less obviously derived characteristics of the logical and eventually, physical designs.

8.1.1. Anomalies

Firstly, swing moorings were assigned a separate table from the other types of berths—the reason being the data that was required to be stored for each was so different, and the prototype wasn't actually required to handle leases for them.

Secondly, *Notes* fields and *Complete* fields were added to several tables simply for the sake of the client, so they could keep track of any extra information (possibly to work around limitations of the system), and to manage incomplete information.

The preliminary logical database design (which can be seen in full in Appendix B) was one of the first designs created for the proposed system. At that stage little regard was being paid to how the database would ultimately relate to the user interface.

Several major difficulties in the business rules manifest themselves in the preliminary logical database design. What follows is a description of these difficulties and how attempts were made to ameliorate them in the

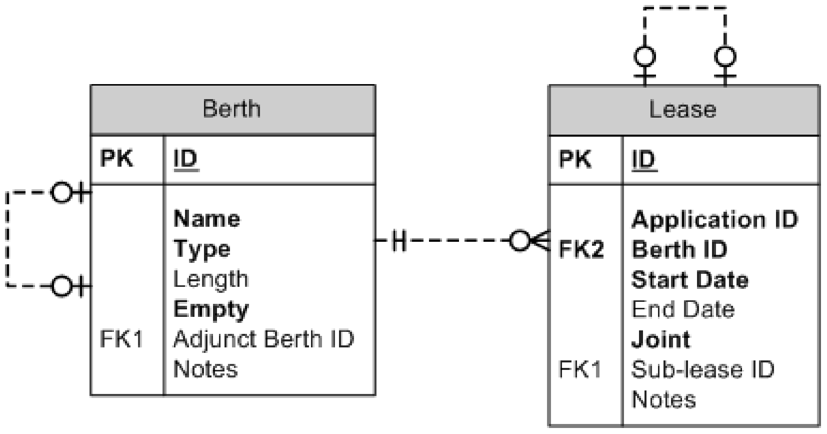


Figure 8.1: The *Berth* and *Lease* entities in the preliminary logical design

final physical design (which can be seen in Appendix D).

8.1.2. Problems

Firstly, the handling of subleasing posed a challenge. There had to be some means for a lease to be somehow linked to another lease, so making it possible for the client to navigate from one to the other, keep track of vacancies, and credit subleased berths.

Secondly, there needed to be a way for, in a sense, two leases to be treated as one. This is to allow for circumstances where a vessel occupies two adjoining berths on the end of a pontoon, but gets treated as a single lease with an extra large berth and vessel.

The preliminary logical design attempted to solve both these problems through the use of recursive relationships (shown in Figure 8.1), so that any given lease can be linked to the relevant sublease in the same entity, and any given berth can be linked to one adjunct berth. The system would then know whether a lease was in fact a joint lease of joint berths, through the use of a *Joint* flag in the *Lease* entity.

The unfortunate thing is that this convenient use of recursion doesn’t account for difficulties handling recursion in the user

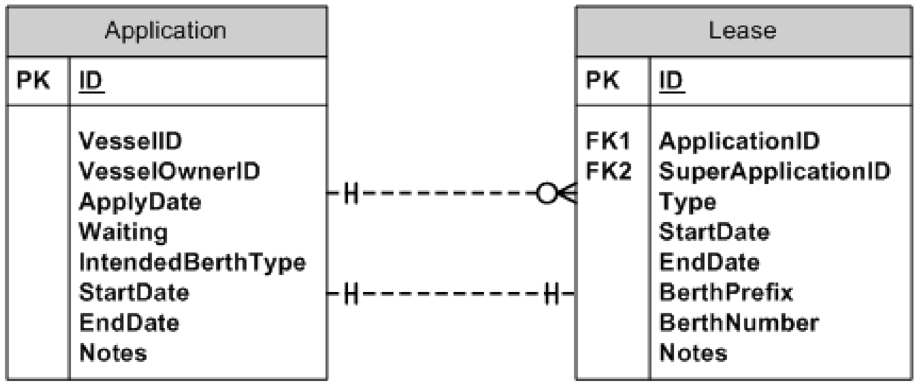


Figure 8.2: The *Application* and *Lease* tables in the final physical design

interface, and complexities with regard to querying and updating records—how will the recursive relationships be traversed efficiently, and where will the recursion end?

Adding the problems with recursion was the fact that a lot of other information needed to be associated with any given lease, including information about the relevant vessel, vessel owner, payments, and temporary absences.

A balance needed to be found between manageability and efficiency. A smaller set of tables would be simpler and more efficient from the perspective of the system, but would require massive redundancy. On the other hand, if the database was made of a large set of inter-connected tables, querying and updating would be very complex and inefficient.

8.1.3. Solutions

The preliminary logical design already accounted for the inter-connectedness of the data by including an *Application* entity which bound vessels, vessel owners and leases together. The next logical step was to make the entity totally central to the database.

As can be seen in Figure 8.2, the recursive *Lease* relationship from the preliminary logical design was replaced with a double relationship in the final physical design. Due to the nature of the majority of queries and updates, the relationship was in fact reversed from the logical to the physical designs—subleases refer to super-leases, instead of the other way round. This relationship is indirect, going through the relevant *Application* records.

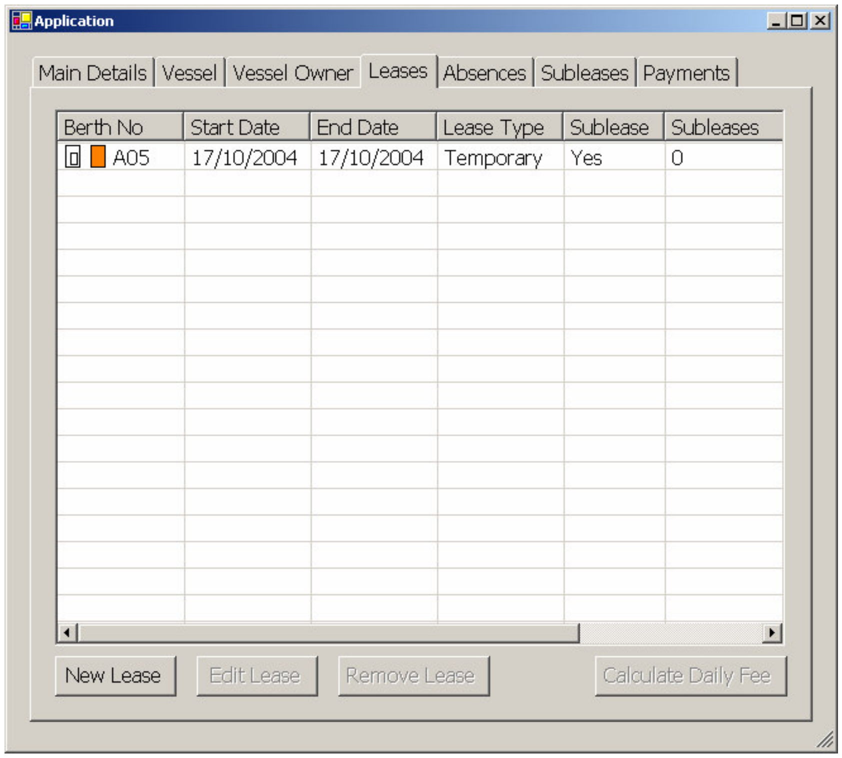
This solution could not be applied to the situation with adjunct berths, for obvious reasons. For this the author decided that the business rules needed to be slightly less strictly interpreted—though in most cases you would not have two leases for the same application at the same time, the system would be able to handle two leases at the same time. The user interface would make it apparent when an application had two leases at the same time.

The final designs' incarnation of the *Application* table also links to all of the tables which were formerly associated with the *Lease* table, so that all queries and updates essentially have to go through the *Application* table. This led to simpler queries and a more effective user interface.

8.2. Interface Design

In the early phases of design, the user interface basically had no depth. Due to the nature of the database and data, the earliest version featured one main form with many tabs on it, each tab featuring a view of the data in the system. All data was viewable from that main form and the only other forms in the design were for editing data.

Before moving on, it is worth noting that, a number of features (mainly print-outs) were never started due to lack of time, and thus never made it into the final design. At no stage during the course of design were the designs truly feature-complete, so even the early incarnations of the user interface were missing visible evidence of such uncompleted features, though locations for them were mentally earmarked.



Berth No	Start Date	End Date	Lease Type	Sublease	Subleases
A05	17/10/2004	17/10/2004	Temporary	Yes	0

Figure 8.3: The *Application* form with the *Leases* tab page visible

8.2.1. Application Form

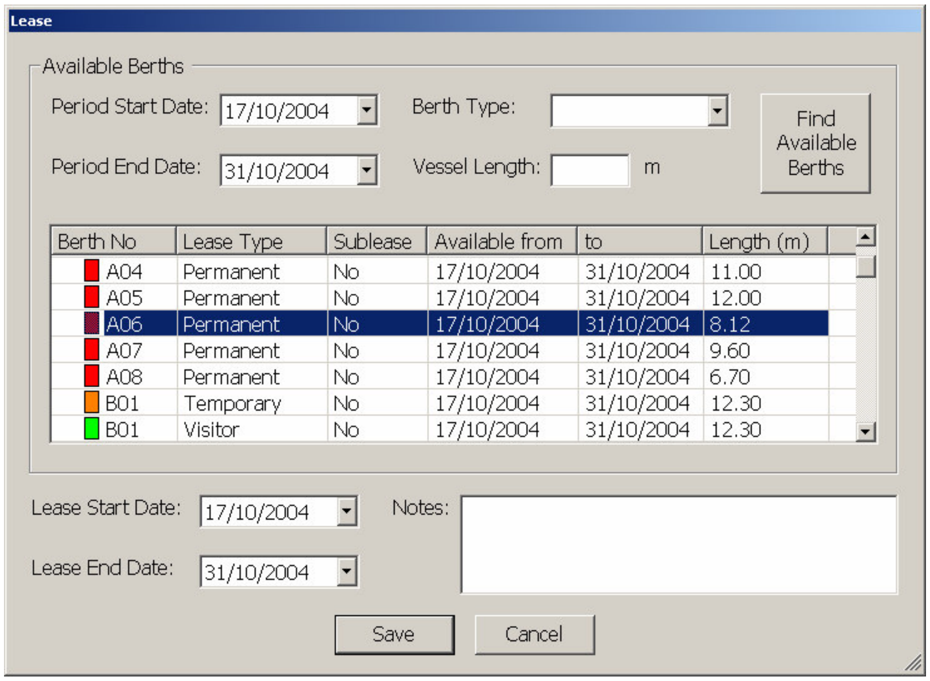
The final user interface improves on the shallow early designs significantly, but in some ways is still rather monolithic. The whole interface is modal in design and hinges around an *Application* dialog (Figure 8.3), which is used to view and enter data for applications. While not the main form, it is the most hard-working part of the interface.

From the *Application* dialog, all data associated with a particular application can be accessed. Leases, temporary absences and payments can be added, edited and removed; vessel and vessel owner information can be selected from the database and edited; and sublease information can viewed and edited.

To add and edit leases the *Application* dialog launches the *Lease* dialog (Figure 8.4). There the user can search for available berths according to specified criteria. Periods of berth availability are then listed. To prevent the user from creating an invalid lease in the view of the business rules, each berth is listed once for each combination of lease type and period of availability that is valid. The user must select one and specify the exact start and end dates for the new lease, while being prevented from entering dates invalid for the selected available berth.

To preserve the integrity of the system's berth bookings when the user edits a lease with the same form, they are prevented from entering an earlier start date or a later end date. The system also takes into account dependent subleases, preventing the user from changing the start and end dates in such a way as to make the subleases invalid. Note that a column showing the actual number of dependent subleases can actually be seen in Figure 8.3.

This principle of guarding against the editing



Period Start Date: 17/10/2004
Period End Date: 31/10/2004
Berth Type:
Vessel Length: m

Berth No	Lease Type	Sublease	Available from	to	Length (m)
🟥 A04	Permanent	No	17/10/2004	31/10/2004	11.00
🟥 A05	Permanent	No	17/10/2004	31/10/2004	12.00
🟥 A06	Permanent	No	17/10/2004	31/10/2004	8.12
🟥 A07	Permanent	No	17/10/2004	31/10/2004	9.60
🟥 A08	Permanent	No	17/10/2004	31/10/2004	6.70
🟧 B01	Temporary	No	17/10/2004	31/10/2004	12.30
🟩 B01	Visitor	No	17/10/2004	31/10/2004	12.30

Lease Start Date: 17/10/2004
Lease End Date: 31/10/2004
Notes:

Figure 8.4: The *Lease* form

of lease periods in such a way as to make subleases invalid also applies to the temporary absences. The user can create and edit these absences on a separate tab page of the *Application* form, but they cannot invalidate subleases which take advantage of a given absence. As with the leases, the absences are each listed with the number of dependent subleases.

Payments (including credits and any fees owing) are listed on yet another tab page of the *Application* form. The user can add, edit and delete these manually; they can also select one of the leases and click a button to automatically calculate leases for a given lease and time period. When a lease with associated payments is deleted, so too are the payments for that lease *if* they are unpaid.

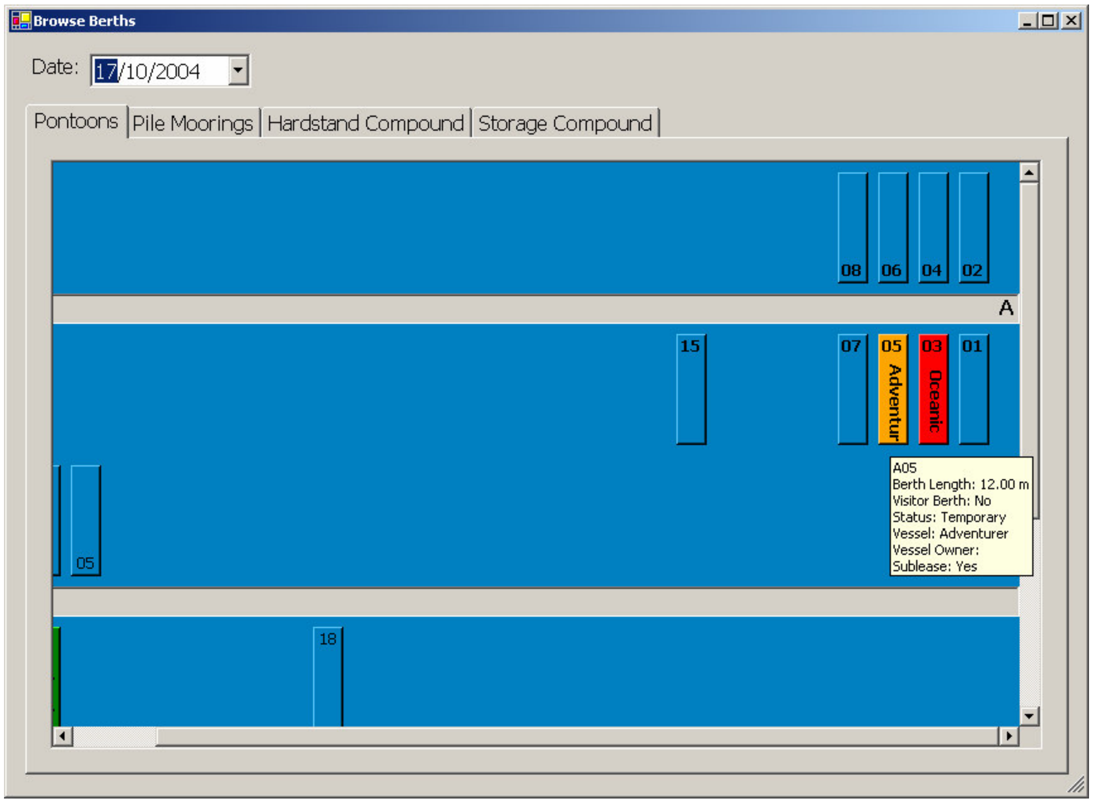
Taking into account the aforementioned database design links subleases to their parents through their applications, the *Application* form allows the user to view and edit the associated applications of dependent subleases. For this it launches another instance of the *Application* form for the relevant sublease application.

8.2.2. View Forms

As mentioned earlier, it was intended that the *Application* form be the focal point for the entire system, and thus be accessible anywhere anything to do with applications was referenced. In the final design this occurs in several places.

Firstly, there is the *Applications* (plural) form, which simply lists all of the applications in the system.

There is also the *Leases* form which instead lists all of the leases (including subleases) in the system. The user can select a lease and be taken to the relevant application. It is



A05
Berth Length: 12.00 m
Visitor Berth: No
Status: Temporary
Vessel: Adventurer
Vessel Owner:
Sublease: Yes

Figure 8.5: The *Browse Berths* form with *PontoonView* custom control

also possible for the user to filter the list by vessel or vessel owner.

Finally, there is the *Browse Berths* form (Figure 8.5), which gives a snapshot of all the berths at a specified point in time—by default, the current date. Pontoon berths are graphically displayed with informational tool tips, while other types of berths are simply listed with their status. The user can select an occupied berth and bring up the relevant application.

8.2.3. Reports Form

A *Reports* form (Figure 8.6) allows the user to create and print graphs and reports. In the final design this is limited to graphs of monthly lease numbers and listings of vessels berthed without electrical warrants of fitness. Originally though, this form contained a separate tab page for statistical analyses.

Because the user may need to print graphs and reports for both the past and the present, the interface was designed to allow the user to specify both the date and the relevant lease type.

Graphs are printed exactly as they are shown on the form apart from being scaled to the page. The listings of vessels without electrical warrants of fitness are presented on the screen as a type of scrollable list view, and printed slightly differently.

To take into account the fact that the listings in their entirety normally wouldn't fit on a single page, the system automatically

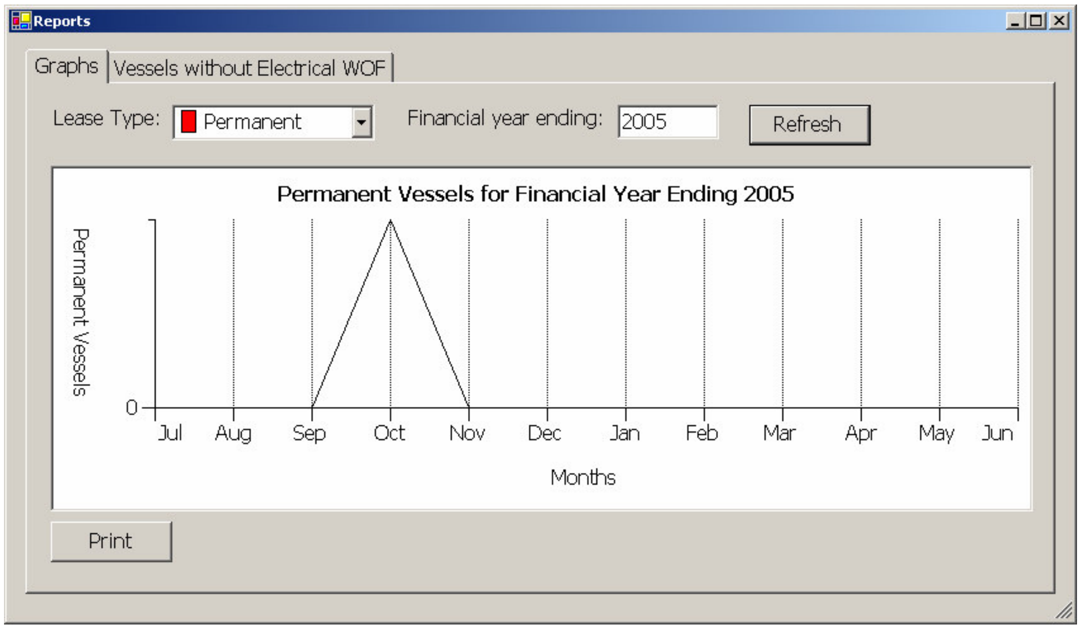


Figure 8.6: The *Reports* form with *Graphs* tab page and *LineGraph* control visible

divides the listing up into pages. Common to each page is a header describing the content of the report, and column headers for the rows of data. The system also prints the page number at the bottom of each page.

8.2.4. Setup Forms

The user is able (nay, required, as no berths come with the system) to create and edit berths of the various types. Berths can also be removed, but for the sake of data integrity the system prevents the user from removing berths which have leases associated with them.

Leasing rates can also be set by the user. For consistency, it is clearly indicated to the user that the all of the rates exclude GST, and the system takes that fact into account.

8.2.5. Backup and Data Movement Features

To ensure that the user does not lose their data in the event of a minor catastrophe, the interface provides functionality from the main window to backup and restore the entire database.

As an added feature, to make changes to the data easily portable, the interface asks at start-up whether or not the user wants to save to file any changes they make during that session. If they answer yes, the interface prompts them to save the changes to a file when they close the program. Copies of the system running on other computers can then be updated from the saved file by using a button on the main form.

As a final note, functionality was added to the main form for destroying and recreating the database. This was mainly for convenience during development. The user is asked for confirmation before any action is taken.

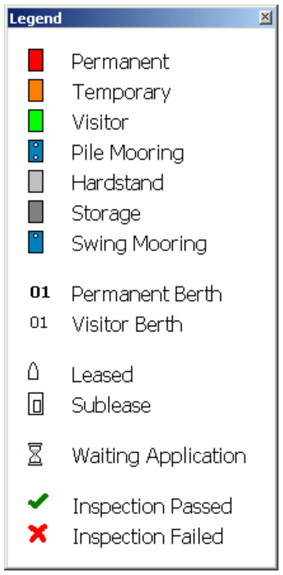


Figure 8.7: The *Legend* form showing the graphical code used

8.2.6. Graphic Design

In the name of ease of use, a colour graphical code was developed to represent the different types of leases, as well as the states of applications and other attributes of records. This code was derived and developed from the original graphical display of berths in the *Browse Berths* form (Figure 8.5).

This code can be seen in the *Legend* form (Figure 8.7), which is the only form in the final prototype that is not modal. The user can summon the legend from the main form. It then stays topmost and translucent while tasks are carried out using the other forms in the system.

8.3. Class Design

Early in the software design process it was decided that the system should be divided into three layers, packages or namespaces: a data access layer, a business logic layer, and a user interface layer. The rationale behind this was to achieve the author's personal goal of a cross-platform design (4.2), as well as to make the classes as manageable as possible.

In addition to this, another package was designed (called *MyLibrary*). This was not dependent on the other classes, and was designed to hold classes which weren't specific to this particular problem. This was done so that they could be reused in future applications, so meeting the author's personal goal of reusability (4.3).

As can be seen in the class designs (Appendix F), all of the top-level packages in the design contain sub-packages which further delineate the classes into dependent groups.

What follows is some highlights of the class designs and the process by which they were derived.

8.3.1. Form Hierarchy

The final class design uses a basic hierarchy of classes for the forms in the system. The only form classes of real note are the *RecordDialog* and *LookupDialog* classes.

The *RecordDialog* form class provides the basic functionality for a form which is used in the process of editing new or existing records in the database. Its descendents contain the functionality to display and extract records from the actual user interface of the form, while notifying the user of any validation errors. Examples of *RecordDialog* descendants include the aforementioned *Application* and *Lease* forms.

The *LookupDialog* form class provides the functionality for a form which is used to prompt the user to select a record from a list. Descendants contain specific functionality to display the different types of record.

8.3.2. Custom Controls

It was immediately clear at the start of the project work that at least one custom control needed to be created. That control is the *PontoonView* control (visible in Figure 8.5), which was required to display the pontoon berths and their statuses.

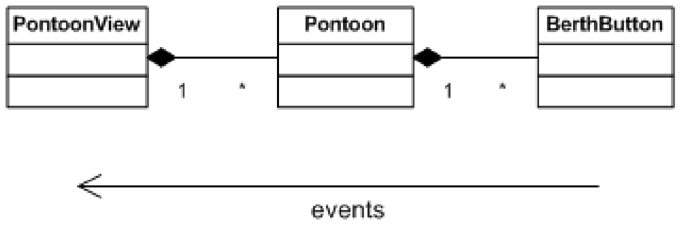


Figure 8.8: The relationship between the *PontoonView*, *Pontoon* and *BerthButton* custom control classes

Behind the scenes, not just one but three classes are used: the *PontoonView* control itself, a *Pontoon* control, and a *BerthButton* control. These are nested inside each other to make up the final display, with click events being propagated up from the *BerthButton* controls to the *PontoonView* control for the sake of simplicity (Figure 8.8).

To save repeating a lot of the same basic code to deal with list views and such things as the facilities to edit records, two control classes were added. These were the *EditListView* and *RecordListView* classes. Both of these classes fire and respond to events.

The *EditListView* control ties a list control with buttons for manipulating the items, so that the enabled state of the buttons automatically changes as items are selected or deselected. Unlike the *RecordListView* class, this class was separated into the *MyLibrary* package.

The *RecordListView* control is a descendant of the *EditListView* control and adds the ability to deal with the list items as records.

For the reporting features, a custom control class (*LineGraph*) to display and print line graphs was also designed. This was added to the *MyLibrary* package. An ancestral *Graph* class was also added to allow for future development (4.3).

Lastly, though not strictly speaking a custom control, a *PrintableTable* class was added for printing the multi-page tables described in 8.2.3.

8.3.3. Design Patterns

Limited research was carried out in the area of object-oriented design patterns for the sake of the data access and business logic packages. A variety of standard design patterns were looked at, but really only one—the Proxy pattern[12]—was found to be applicable; however, it was ultimately never used.

The following design for the data access framework was largely inspired by an approach used in a commercial content management system[13].

8.3.4. Data Access

In comparison to the other layers, the data access layer contains few classes. The main ones are the *Database* class, *Record* class, and *Function* class.

Before continuing, it must be noted again that a major personal goal was to end up with a design that was as cross-platform as possible (4.2).

The *Database* class is essentially the embodiment of the database connection. It is responsible for creating, backing up and restoring the database. Queries and updates are also handled by this class, which for efficiency are all done with SQL.

[12] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, USA.

[13] Spolsky, J. (2003). *Joel on Software – CityDesk Entity Classes.* Retrieved 10 August, 2004 from http://www.joelonsoftware.com/articles/CityDeskEntityClasses.html.

The abstract *Record* class provides the functionality to read fields off a row returned by an instance of the *Database* class in response to a query. It can also add or update a row in a table. Descendants implement the abstract functionality for specific tables, with the instances acting as type-safe conduits for data in the database and providing basic integrity checks.

The abstract *Function* class actually links the *Database* and *Record* classes together. It takes a query, executes it against a *Database* object and then creates and reads in instances of the appropriate *Record* subclasses to suit. While doing this, it takes into account the different tables that are joint in the query.

8.3.5. SQL Handling

To the end of cross-platform design and manageability, it was a intended if at all possible the final implementation would not mix SQL statements and code. It was desired that the design take advantage of the string formatting routines common to many programming languages, whereby special markers in strings can be replaced with substrings at runtime.

Two other classes in the data access package (*SqlResource* and *SqlLog*) allow the loading of groups of SQL statements from text resources, and the saving of groups of SQL statements to text files. This is for the purposes of separating SQL from code, and saving logs of changes to the database, respectively. As they share an inheritance bond, the classes both use the database-specific batch separators to break the groups of statements up into batches. This makes it easy to switch between a query analyser and the development environment.

The *SqlResource* class can also extract names for the SQL batches which have been specified in the comments of the SQL. It was designed this way so that code can reference a particular SQL statement or statements by name, and so the order of the batches of statements in the resource doesn't matter.

8.3.6. Business Logic

The business logic is by far the largest package in the project, which is unsurprising given the complexity of the business rules. Virtually all classes in it are descendants of the above-mentioned *Function* class, or its close descendent in the logic layer, the *RecordList* class, which adds record creation and updating functionality—with checks for data integrity.

Basic descendents of the *Function* class implement the functionality needed to retrieve and filter the appropriate data for the view forms of the user interface, while the complex *Application* form is dependent on a close-knit sub-package of related *RecordList* descendents.

To generate a view of the marina for a specific date for the *Browse Berths* form, and to extract the available berths for the *Lease* form, a *BerthPeriodView* class was included. This class cross-references lease, absence and berth records to generate a snapshot of the berths over a given period of time. For the *Browse Berths* form this period is one day.

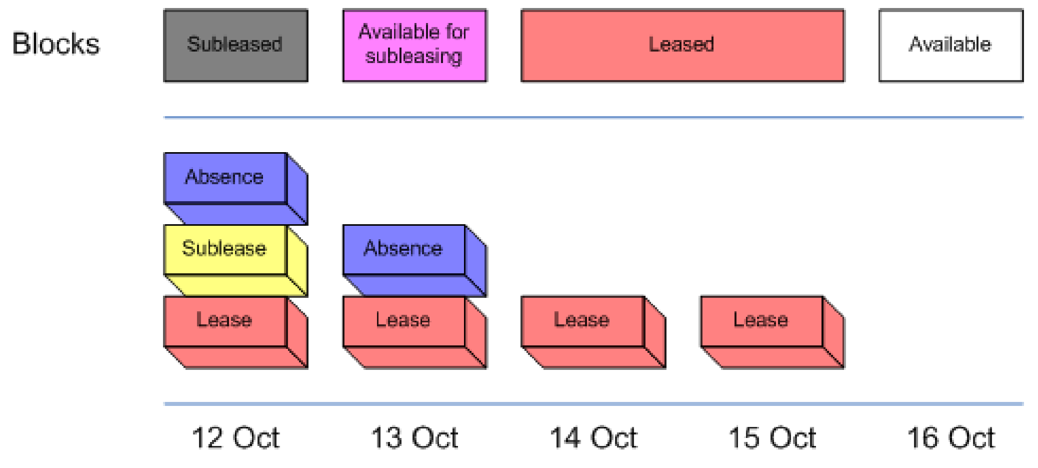


Figure 8.9: An illustration of the inner workings of the *BerthPeriodView* class

The technique used to generate this snapshot is moderately complex, but can be analogised as layered pieces of coloured glass (seen in Figure 8.9).

With the help of supporting classes, for each berth the *BerthPeriodView* class can be imagined to lay a piece of coloured glass on a calendar for each day the berth is leased. It then goes through all the absences and overlays differently "coloured pieces of glass" on the calendar for each of these days. When finished, it scans through the calendar for contiguous blocks of "colour" generated by the layered pieces of "coloured glass".

Descendents of the *BerthPeriodView* class take the generated segments of "colour" and extract periods of availability for berths or berth statuses, as the need dictates.

Finally, the functionality to extract the monthly totals of vessels for the graphing facilities in the *Reports* form, is included in a *VesselMonthReport* class.

9. Implementation

Implementation, as the reader may have inferred, was started partway through the design process. This section mainly serves as a discussion of the stickier platform implementation issues.

9.1. Platform Selection

Because of the cross-platform intent of the author, the selection of appropriate platform for the final prototype was left till this late stage. As mentioned in earlier sections, there was some ambiguity over how accessible and portable the data needed to be, which led to the consideration of web-based solutions.

ASP.NET (6.3) was considered for its ease of use, power, and potential. The author had some cursory experience with it, and, as it was an emerging technology, wished to acquire a deeper knowledge. This included gaining experience with the C# language, which is part of ASP.NET and was known to be similar to C++, a favourite language of the author's.

The other, more common and freely available scripting languages such as Perl, Python and PHP were considered, but were judged to be too unfamiliar in syntax and architecture.

What ultimately led to a re-evaluation of the destination environment was a true realisation of the high costs involved in ASP.NET hosting. For the sake of the client it had always been the intent to make the solution as cheap as possible, so in discussions with the client the system requirements were reconsidered. As it turned out, a desktop solution was adequate for the task.

Thankfully, the author did not need to develop the prototype under ASP.NET to gain the desired experience with the C# cross-platform programming language; C# can be used in the development of desktop applications. As such, it became the development platform, with Microsoft Visual Studio serving as the integrated development environment.

9.1.1. Database

The choice of database management system (DBMS) was somewhat less important than the choice of platform, due to the fact that there are a small number of ubiquitous cross-platform database connectivity standards. Three DBMSs were considered, in turn.

Firstly, the MySQL DBMS[14] was considered for its adherence to the ISO SQL standards and open source, cross-platform nature. However, it was eventually rejected due to the confusing licensing terms, which might have required to client to spend a lot of money.

Microsoft's Jet DBMS was considered next for its ubiquity across the Windows platform, and for the fact it is free to use. It was rejected early in the implementation because of problems getting it to work, and a near-complete lack of documentation.

Microsoft's SQL Server Desktop Engine[15] was finally chosen for its wide-spread support and easy licensing terms.

9.2. Platform Adaptations

In the process of implementation, certain adaptations were required to tailor the design to the chosen platform. These adaptations included making classes inherit from the .NET classes, and using non-standard SQL script dialects. The most notable, yet least obvious adaptations will now be described.

---

[14] *MySQL: The World's Most Popular Open Source Database.* Retrieved 17 July, 2004 from http://www.mysql.com.

[15] *Microsoft SQL Server: MSDE 2000 Home.* Retrieved 25 July, 2004 from http://www.microsoft.com/sql/msde/default.asp.

9.2.1. Data Access

In the .NET Framework, database queries return instances of classes which implement the *IDataReader* interface. This interface provides the means to iterate through the returned rows of a query. It descends from the *IDataRecord* interface, which contains accessors for fields. To make it possible for the *Function* class to generate *Record* instances corresponding to the rows of a given query, the *Database* class returns an *IDataReader* for each query, and the Record class reads in data from an *IDataRecord*. This is illustrated in Figure 9.1.

9.2.2. Custom Controls

To get the appropriate bordered look of the *PontoonView* (Figure 8.5) and *LineGraph* (Figure 8.6) control classes, they actually had to inherit from the .NET *Panel* class, rather than the expected *ScrollableControl* and *Control* classes, respectively.

Continuing with the *LineGraph* class, the relevant code took advantage of the .NET Framework which features a *Graphics* class for drawing, instances of which can represent the screen or the printer. Taking advantage of this fact, the *LineGraph* class

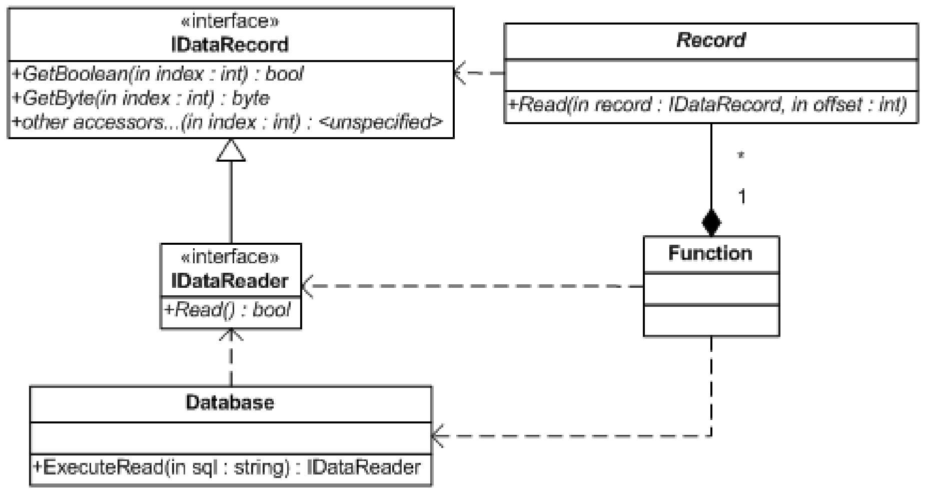

«interface»
IDataRecord
+GetBoolean(in index : int) : bool
+GetByte(in index : int) : byte
+other accessors...(in index : int) : <unspecified>
Record
+Read(in record : IDataRecord, in offset : int)
*
1
«interface»
IDataReader
+Read() : bool
Function
Database
+ExecuteRead(in sql : string) : IDataReader

Figure 9.1: The platform data access adaptations made regarding the *IDataRecord* and *IDataReader* interfaces

actually uses exactly the same code for both drawing the graph on screen and drawing the graph on a printer page.

9.3. Debugging

In addition to the usual set of debugging tools that the development environment had to offer, several techniques were developed to handle the task.

Firstly, as a result of the fact that the system was a database application, an external query analyser played a large role in debugging. In the process of fixing bugs, the author switched between the development environment's debugger and the query analyser constantly, to eliminate SQL queries as a source of problems.

On the opposite side of things, the custom control classes and classes for printing graphs and tables were barraged with an array of fabricated data before they were ever made data-aware, to rule them out as a source of problems.

Lastly, assertions were used to perform checks for the correctness of parameters in methods, as well basic integrity checks on data pulled in from the database and resources.

9.4. Testing

Though no formal tests were carried out, testing (including those above) was performed throughout implementation to ensure the parts being worked on operated correctly.

During demonstrations of prototypes, any issues that came up were also noted and later corrected, if possible. As noted in 5.1.3, the client was also urged to hold on to and play around with the prototypes, notifying the author of any problems they had.

10. Evaluation

Overall the author was very satisfied with the capabilities and quality of the final prototype, and this was reflected by the client's praise. However, this is not to say there were not things about the project work which the author felt could have gone better.

10.1. Client Relations

Communications with the client formed a significant and pervasive part of the project work. Unfortunately, as noted in previous sections, some misunderstandings arose which showed that the author's interpersonal communication skills could be improved.

Another significant contributor to these misunderstandings was the fact that the client was largely computer illiterate. Describing how not only the prototypes produced, but computers in general worked proved to be arduous and difficult to avoid. However, the client was urged to learn as much as possible in their own time.

10.2. Methodology

In terms of the methodology used, it was felt that the order of the analysis and design phases could have been improved.

The sequence of analysis and design, as expounded in the previous sections, featured database analysis and design first, and then software design. Database analysis and design was carried out using conceptual, logical and physical designs, while software design involved the development of use cases, interface designs and class diagrams.

In the end, the use cases and interface designs developed in the software design phase were found to be more useful in capturing the scope of the business rules than the process of creating database designs had been.

10.3. Cross-platform Focus

It was felt that the cross-platform focus of the design and implementation parts of the project were too ambitious. Many issues cropped up with regard to this aspect which, in hindsight, would have been better served by a project brief with fewer business requirements and more time.

10.4. Cohesion

On the bright side of things, the large number of the classes created for the prototype ended up being very cohesive. Though on the surface the total number of classes in the final prototype may seem excessive (there are around 70 in total), each class served to delineate the code enough to make the complex business rules and functionality manageable.

The separation of the design into layers and the use of packages and sub-packages also provided a beneficial cohesion between groups of classes and of functionality—they made a complex task simpler.

10.5. Reusability

In contrast with the goal of a cross-platform design, the emphasis on reusability with regard to future applications went well. There was no difficulty to be had separating the reusable elements from the non-reusable elements. Not only that, but this weeding out of reusable elements seems to have been very conducive to good design, and, most importantly, the author was left with a very useful array of classes for use in future projects.

The prototype-specific classes were, at least from the perspective of the user interface classes, very reusable too. Apart from the *Application* form, each form class required very little code—only the most basic routines to display data in, and extract data from the user interface were required.

10.6. Platform

By and large, the author was pleased with the platform selected for implementation. Though the C# language had not been used before, it was able to be picked up easily due to its similarity with languages the author had experience with. In fact, it was found to be in many ways superior to these previously experienced languages, due to its balance between cleanliness and power.

10.7. Development Environment

The development environment used (Microsoft Visual Studio .NET) was found to be very flexible and powerful; however, it did have some not-insignificant usability issues when it came to the form designer. The most verifiably incorrect of these behaviours was a chronic inability to save forms correctly.

Under Visual Studio .NET, the form designer saves the controls, their properties and layout as initialisation code in a hidden section of the relevant class source code file. Every time the user builds the application, the development environment refreshes the form layout in the form designer, from the code.

On multiple occasions a change was made to a control on a particular form through the form designer. The application was then rebuilt, and form designer view changed right in front of the author's eyes. To work around these cases, the relevant initialisation code had to be manually edited.

11. Conclusions

As a whole, the project work documented in this report served to demonstrate three key facts about the information technology industry: the value of object-oriented design, the importance of good scope management, and the necessity of training.

11.1. Object-oriented Design

Object-oriented (OO) design has been held as an all-purpose cure for software development woes by many developers, since it was first conceived. While some people may well have overstated the importance of OO design, this project has shown the power of such a methodology.

The mere fact that a complex set of business rules like the marina's were able to be implemented in the relatively short time available speaks volumes. Add to this the amount of functionality achieved, and the cohesive, manageable and reusable fashion in which it was achieved and you have a winning paradigm.

11.2. Scope Management

Though proper OO design can go a long way, it is not a magic bullet. It cannot increase the number of hours in a day, nor can it make complex business rules simple.

It's always good to have high ambitions, but as is often the case with software development—and in particular this project—those ambitions, in conjunction with OO methods, can obscure the inherent complexity of a problem. In the case of this project, more, practical functionality may well have been able to be completed in the time given, if the author hadn't put so as much emphasis on things like cross-platform design.

11.3. Training

Following on, even if you manage to include all the practical features in the world, it doesn't mean a thing if the users don't know how to use them. This applies both to custom software systems such as the project prototype and to information technology in general.

> And in the case of custom software systems, it certainly doesn’t help in development. Much time that could have been spent more constructively on this particular project was wasted explaining what should have been basic things.
>
> You could argue that the purpose of information technology professionals is to absolve the need for users to have any expertise. Realistically though, for the use of information technology to really be effective, computers need to be approached like cars, with regards to training. Plenty of people don’t know how car engines work but everybody who drives has invested significant time in learning how to drive safely and properly.

To sum up: although information technology has such huge potential in so many areas, the failure of it to penetrate some of those areas could probably be blamed on an underestimation of the work required, both on the user’s side of things and on the developer’s.

This particular project, while largely successful in solving the stated problem, could have achieved more had there been a bit more balance between the author’s personal goals and the client’s requirements.

12. Bibliography

- *The Apache Struts Web Application Framework.* Retrieved 15 July, 2004 from http://struts.apache.org.

- *ASP.NET Web: The Official Microsoft ASP.NET Web Site.* Retrieved 17 July, 2004 from http://www.asp.net.

- Connolly, T. & Begg, C. (2002). *Database Systems: A Practical Approach to Design, Implementation, and Management* (3rd ed.). Addison-Wesley.

- *Dock Slips.* Retrieved 11 July, 2004 from http://www.ids-astra.com/dock_slips.

- *eric3.* Retrieved 18 July, 2004 from http://www.die-offenbachs.de/detlev/eric3.html.

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley.

- Hall, M. *Core Servlets and JavaServer Pages.* Retrieved 14 July, 2004 from http://csajsp-chapters.corewebprogramming.com/Core-Servlets-and-JSP.pdf.

- *HavenStar.* Retrieved 11 July, 2004 from http://www.starleisure.com/HSMOverview.asp.

- *IDE.PHP.* Retrieved 18 July, 2004 from http://www.ekenberg.se/php/ide.

- *Java Technology.* Retrieved 14 July, 2004 from http://java.sun.com.

- Laudon, K. C. & Laudon, J. P. (1996). Essentials of Management Information Systems: Organization and Technology. Prentice-Hall.

- *Macromedia ColdFusion MX.* Retrieved 15 July, 2004 from http://www.macromedia.com/software/coldfusion/.

- *Marina Management: marina software, marina management software.* Retrieved 7 July, 2004 from http://www.execu-tech.com/marinas.shtml.

- *Marina Management Software – Marina Manager.* Retrieved 10 July, 2004 from http://www.marina-management-software.com.

- *Marina Management System (MMS).* Retrieved 10 July, 2004 from http://www.pacsoft.co.nz/mms/MMS%20Demo_files/MMS%20Demo.pps.

- *MarinaOffice*. Retrieved 7 July, 2004 from http://www.scribbletraining.com/Merchant2/modemo/modemo.htm.

- *MarinaWare.* Retrieved 11 July, 2004 from http://www.marinaware.com.

- *Microsoft ASP.NET QuickStarts Tutorial.* Retrieved 17 July, 2004 from http://www.dotnetjunkies.com/quickstart/aspplus/doc/applications.aspx.

- *Microsoft SQL Server: MSDE 2000 Home.* Retrieved 25 July, 2004 from http://www.microsoft.com/sql/msde/default.asp.

- *MySQL: The World's Most Popular Open Source Database.* Retrieved 17 July, 2004 from http://www.mysql.com.

- *NetBeans.* Retrieved 14 July, 2004 from http://www.netbeans.org.

- *Ocelot.* Retrieved 10 August, 2004 from http://www.ocelot.ca.

- *Open Perl IDE.* Retrieved 18 July, 2004 from http://open-perl-ide.sourceforge.net.

- *Perl Builder.* Retrieved 18 July, 2004 from http://www.solutionsoft.com/perl.htm.

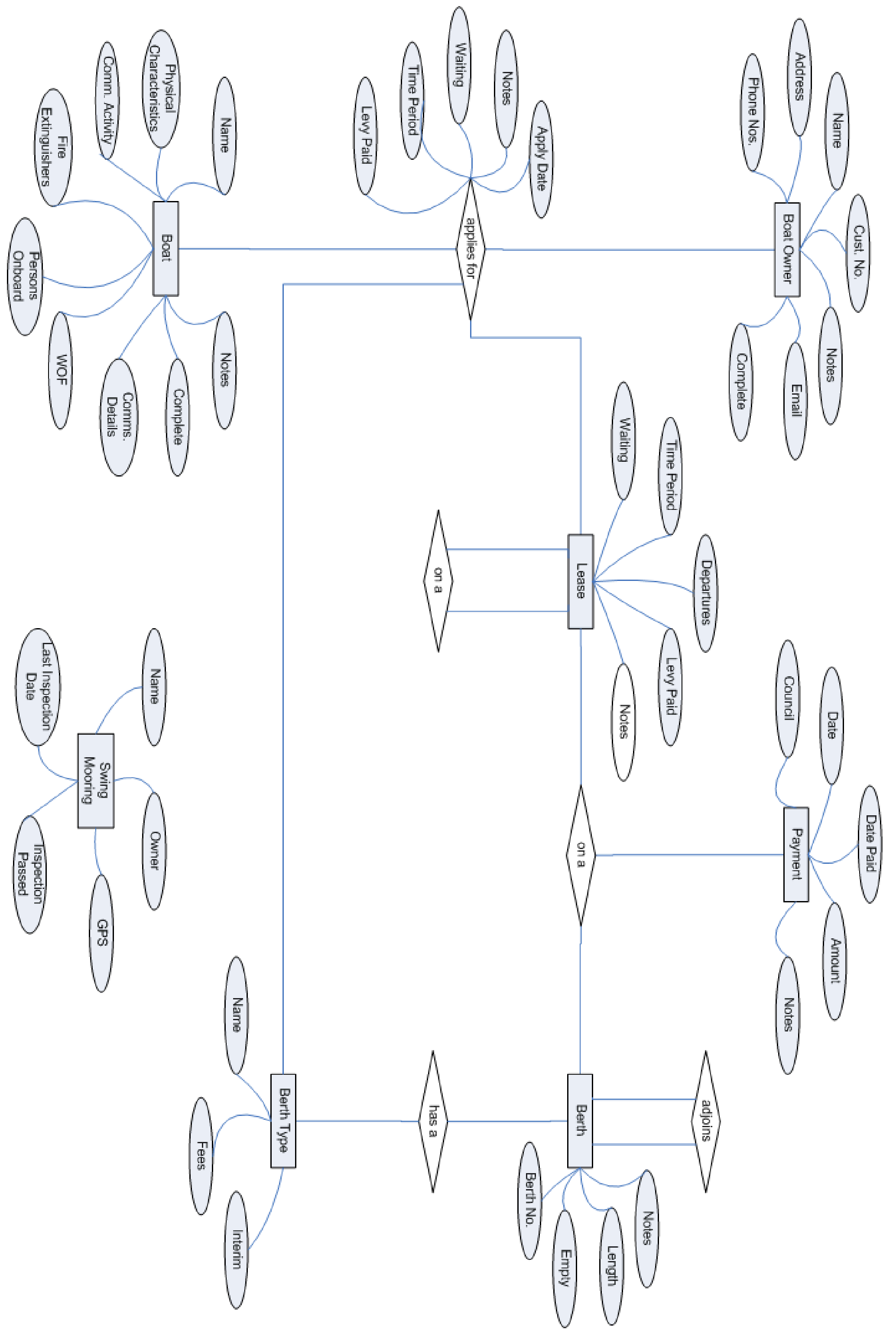
- *The Perl Directory – perl.org.* Retrieved 16 July, 2004 from http://www.perl.org.

- *PhpEd.* Retrieved 18 July, 2004 from http://www.nusphere.com/products/index.htm.

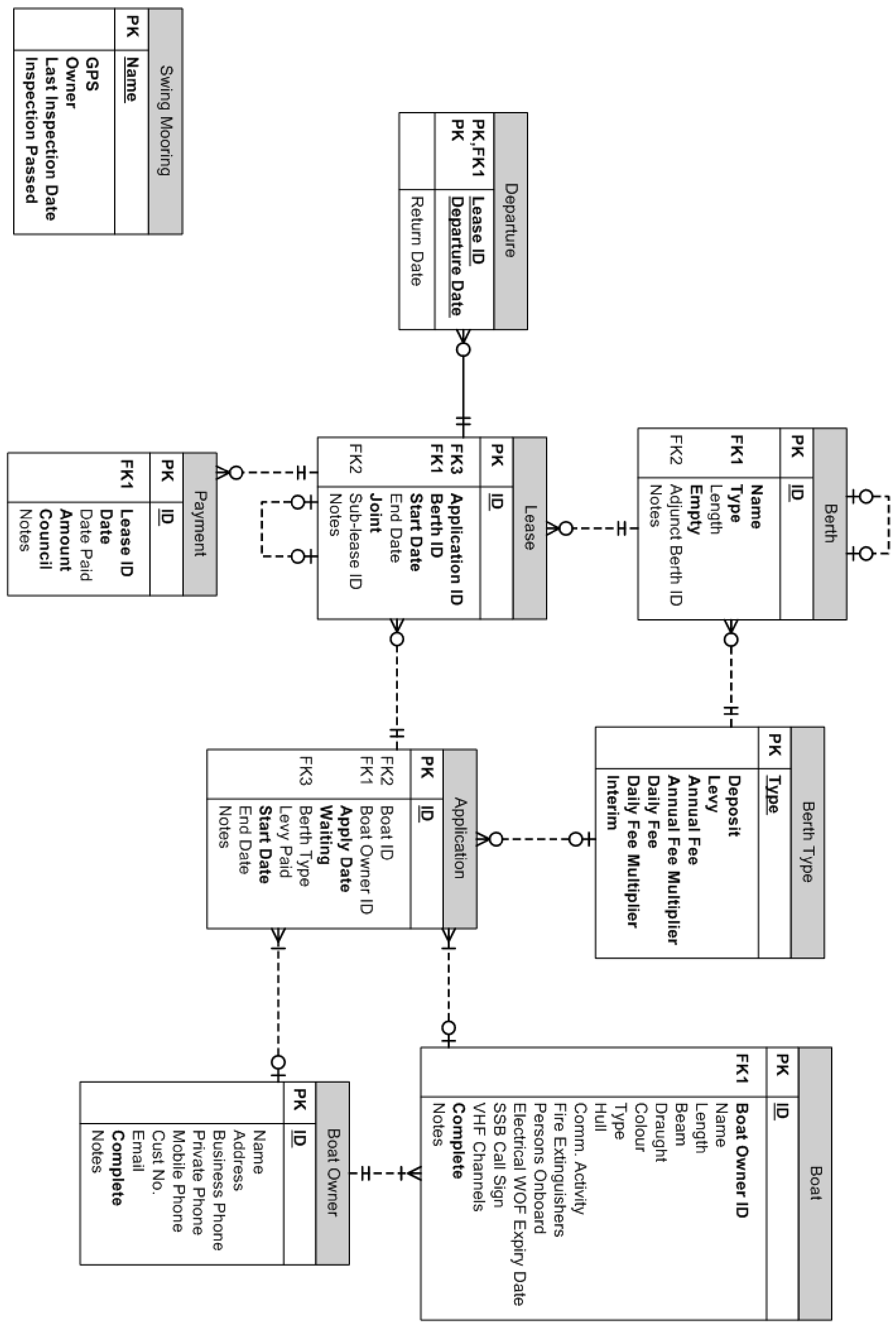
- *PHP: Hypertext Preprocessor.* Retrieved 16 July, 2004 from http://www.php.net.

- *Python Programming Language.* Retrieved 17 July, 2004 from http://www.python.org.
- *Python Tutorial: Python.* Retrieved 17 July, 2004 from http://martin.f2o.org/python/tutorial.
- Remenyi, D. (1993). *Information Management Case Studies.* London, UK: Pitman Publishing.
- *Servlets and JSP: An Overview.* Retrieved 14 July, 2004 from http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/.
- Spolsky, J. (2003). *Joel on Software – CityDesk Entity Classes.* Retrieved 10 August, 2004 from http://www.joelonsoftware.com/articles/CityDeskEntityClasses.html.
- Spolsky, J. (2001). *Joel on Software – User Interface Design For Programmers.* Retrieved 11 August, 2004 from http://www.joelonsoftware.com/uibook/fog0000000249.html.
- VerBeek, T. (2003). *Visual Basic, Active Server Pages, and other web scripting technology.* Retrieved 15 July, 2004 from http://microsoft.toddverbeck.com/script.html.
- *What is Mono?* Retrieved 16 July, 2004 from http://www.mono-project.com.
- *Wing IDE.* Retrieved 18 July, 2004 from http://wingide.com/wingide.
- *Zope.* Retrieved 15 July, 2004 from http://zope.org.

APPENDIX A: Preliminary Conceptual Database Design



APPENDIX B: Preliminary Logical Database Design



APPENDIX C: Preliminary Logical Data Dictionaries

Application

FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Boat ID	Integer	
Boat Owner ID	Integer	
Apply Date	Date	DD/MM/YYYY
Waiting	Boolean	
Berth Type	Text	20 Characters
Levy Paid	Boolean	
Start Date	Date	DD/MM/YY
End Date	Date	DD/MM/YY
Notes	Text	200 Characters

Berth

FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Name	Text	4 Characters
Type	Text	20 Characters
Length	Decimal	4.00-50.00
Empty	Boolean	

FIELD NAME	***TYPE***	***RANGE***
Adjunct Berth ID	Integer	
Notes	Text	200 Characters

Berth Type

FIELD NAME	***TYPE***	***RANGE***
Type	Text	20 Characters
Deposit	Currency	
Levy	Currency	
Annual Fee	Currency	
Annual Fee Multiplier	Currency	
Daily Fee	Currency	
Daily Fee Multiplier	Currency	
Interim	Boolean	

Boat

FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Boat Owner ID	Integer	
Name	Text	30 Characters
Length	Decimal	4.00-50.00

FIELD NAME	***TYPE***	***RANGE***
Beam	Decimal	0.10-10.00
Draught	Decimal	0.10-10.00
Colour		Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black, White
Type		Yacht, Motor sailor, Launch
Hull		Mono, Catamaran, Trimaran
Comm. Activity	Boolean	
Fire Extinguishers	Integer	0-10
Persons Onboard	Integer	0-15
Electrical WOF Expiry Date	Date	DD/MM/YY
SSB Call Sign	Text	10 Characters
VHF Channels	Text	10 Characters
Complete	Boolean	
Notes	Text	200 Characters

Boat Owner

FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Name	Text	50 Characters

FIELD NAME	***TYPE***	***RANGE***
Address	Text	200 Characters
Business Phone	Text	20 Characters
Private Phone	Text	20 Characters
Mobile Phone	Text	20 Characters
Cust. No.	Text	8 Characters
Email	Text	30 Characters
Complete	Boolean	
Notes	Text	200 Characters

Departure

FIELD NAME	***TYPE***	***RANGE***
Lease ID	Integer	
Departure Date	Date	DD/MM/YY
Return Date	Date	DD/MM/YY

Lease

FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Application ID	Integer	

FIELD NAME	***TYPE***	***RANGE***
Berth ID	Integer	
Start Date	Date	DD/MM/YYYY
End Date	Date	DD/MM/YY
Joint	Boolean	
Sub-lease ID	Integer	
Notes	Text	200 Characters

Payment

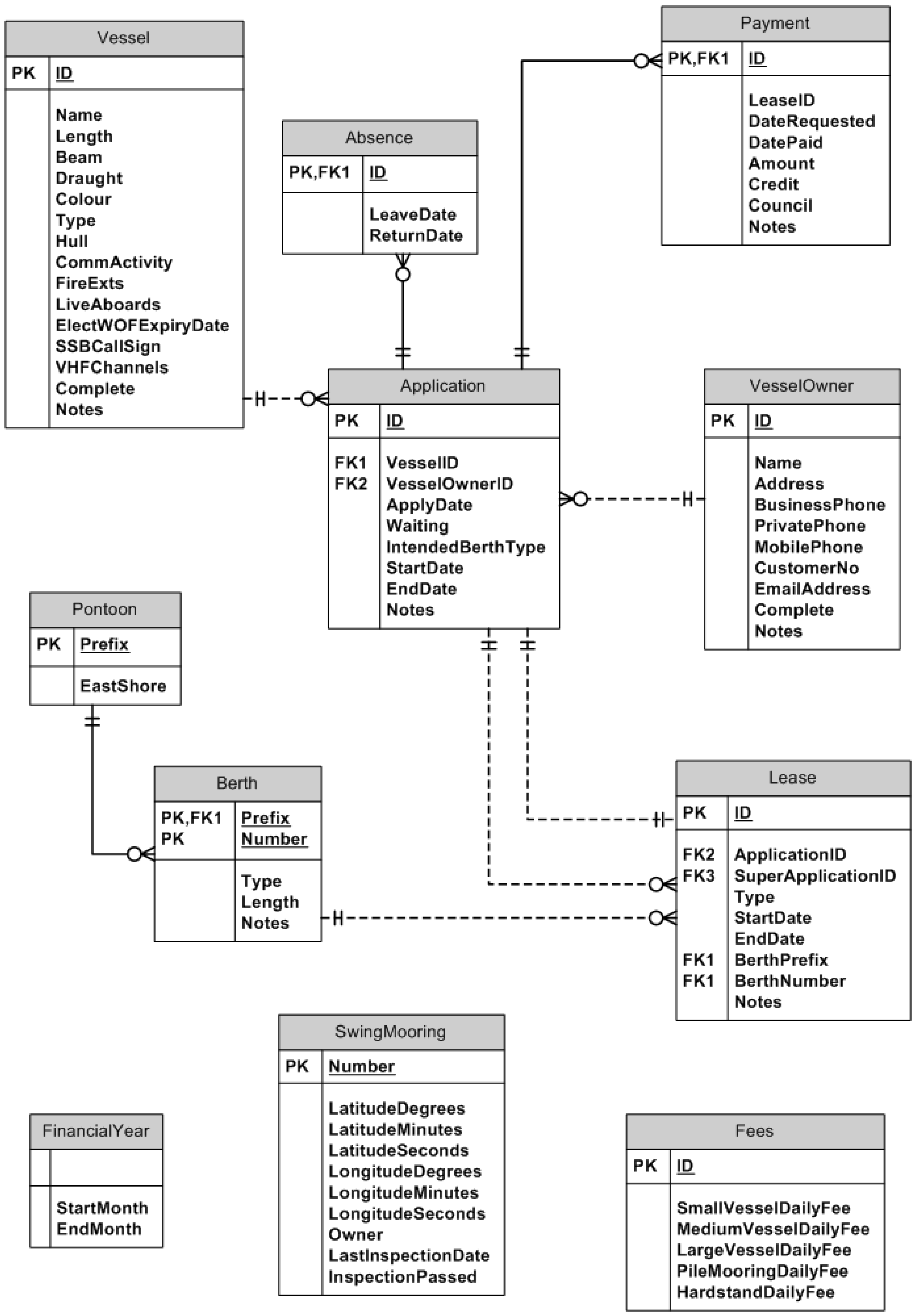
FIELD NAME	***TYPE***	***RANGE***
ID	Integer	
Lease ID	Integer	
Date	Date	DD/MM/YY
Date Paid	Date	DD/MM/YY
Amount	Currency	+/-
Council	Boolean	
Notes	Text	200 Characters

Swing Mooring

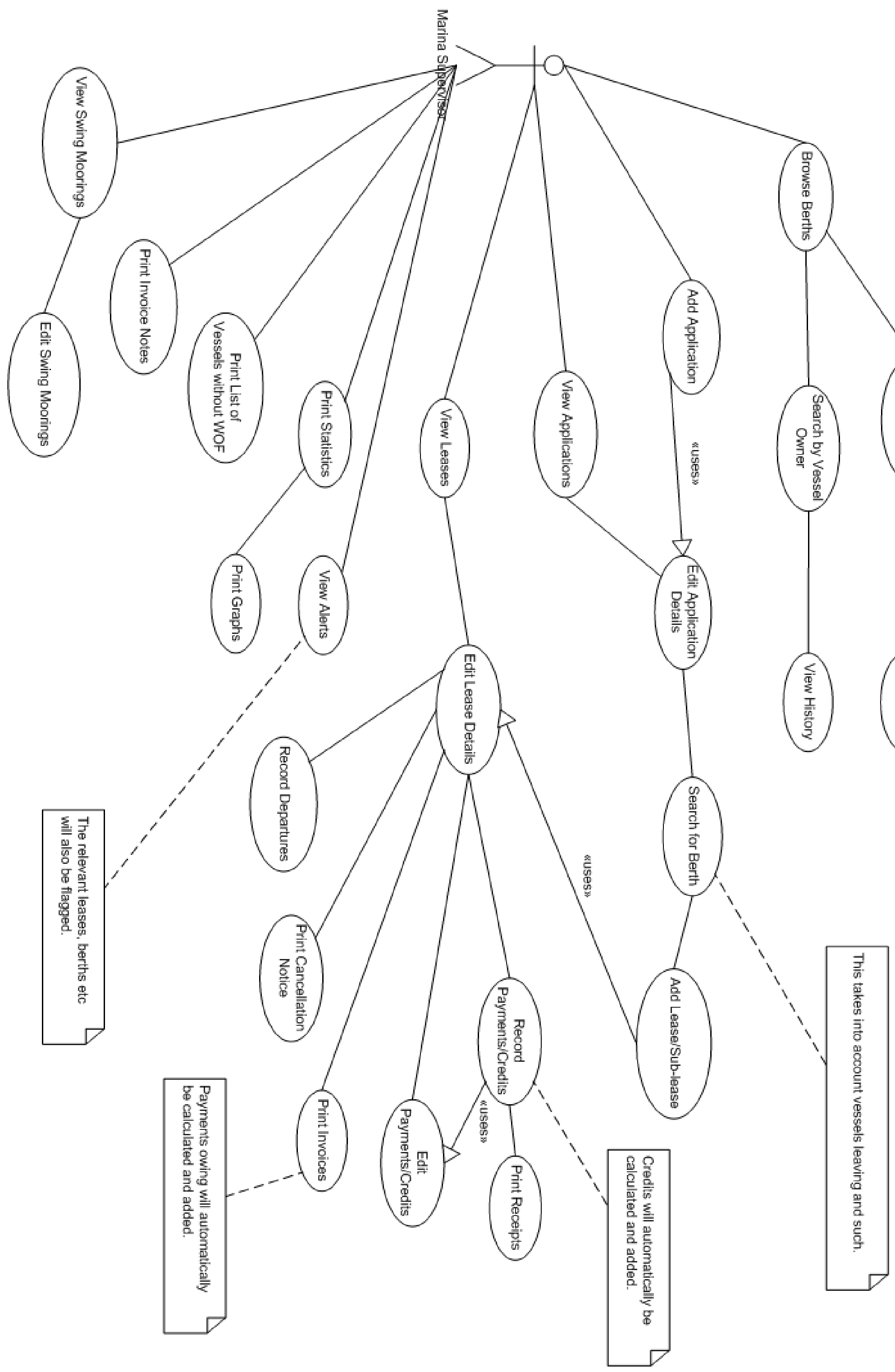
FIELD NAME	***TYPE***	***RANGE***
Name	Text	4 Characters
GPS	Text	10 Characters
Owner	Text	50 Characters

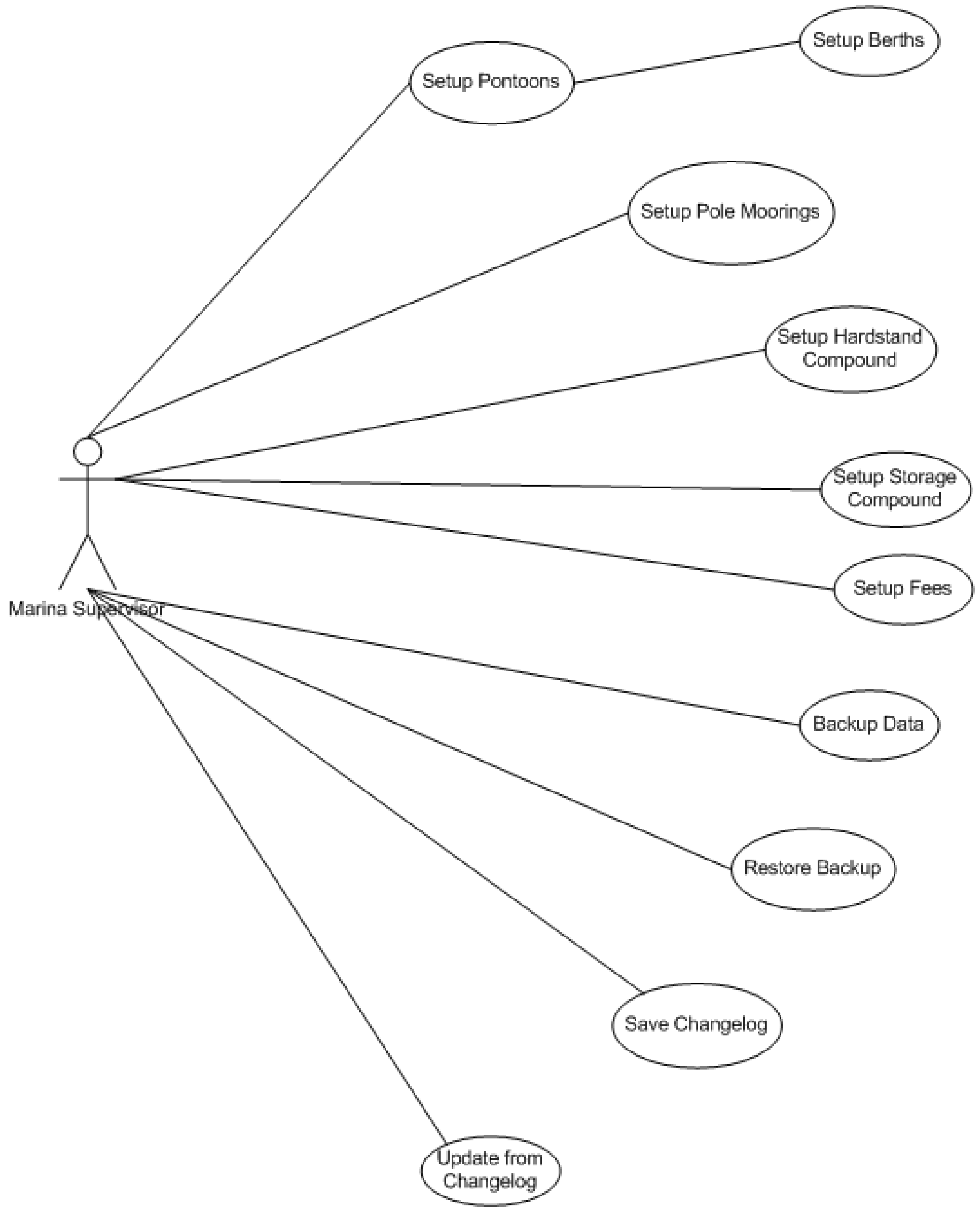
FIELD NAME	***TYPE***	***RANGE***
Last Inspection Date	Date	DD/MM/YY
Inspection Passed	Boolean	

APPENDIX D: Physical Database Design



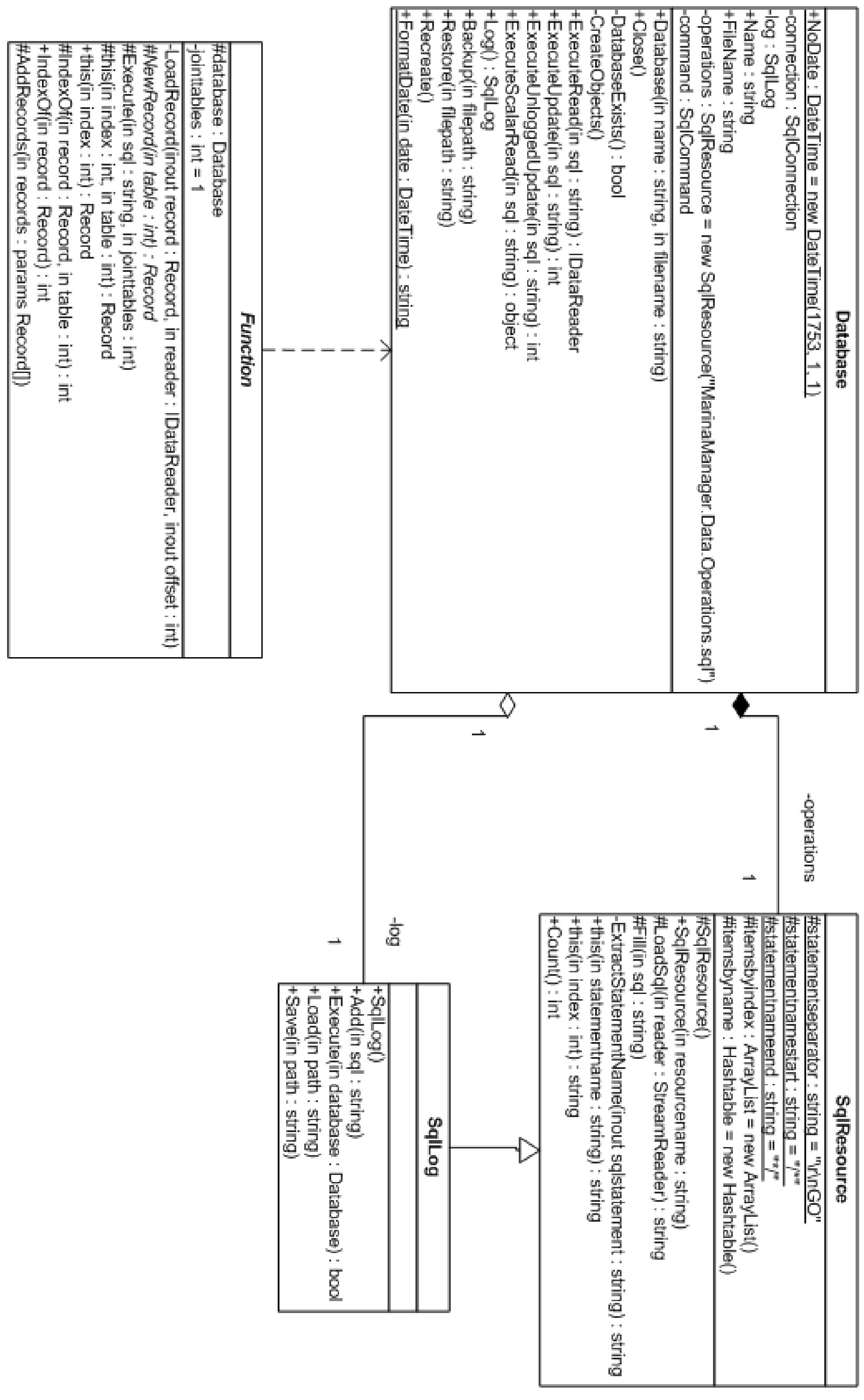
APPENDIX E: Use Cases



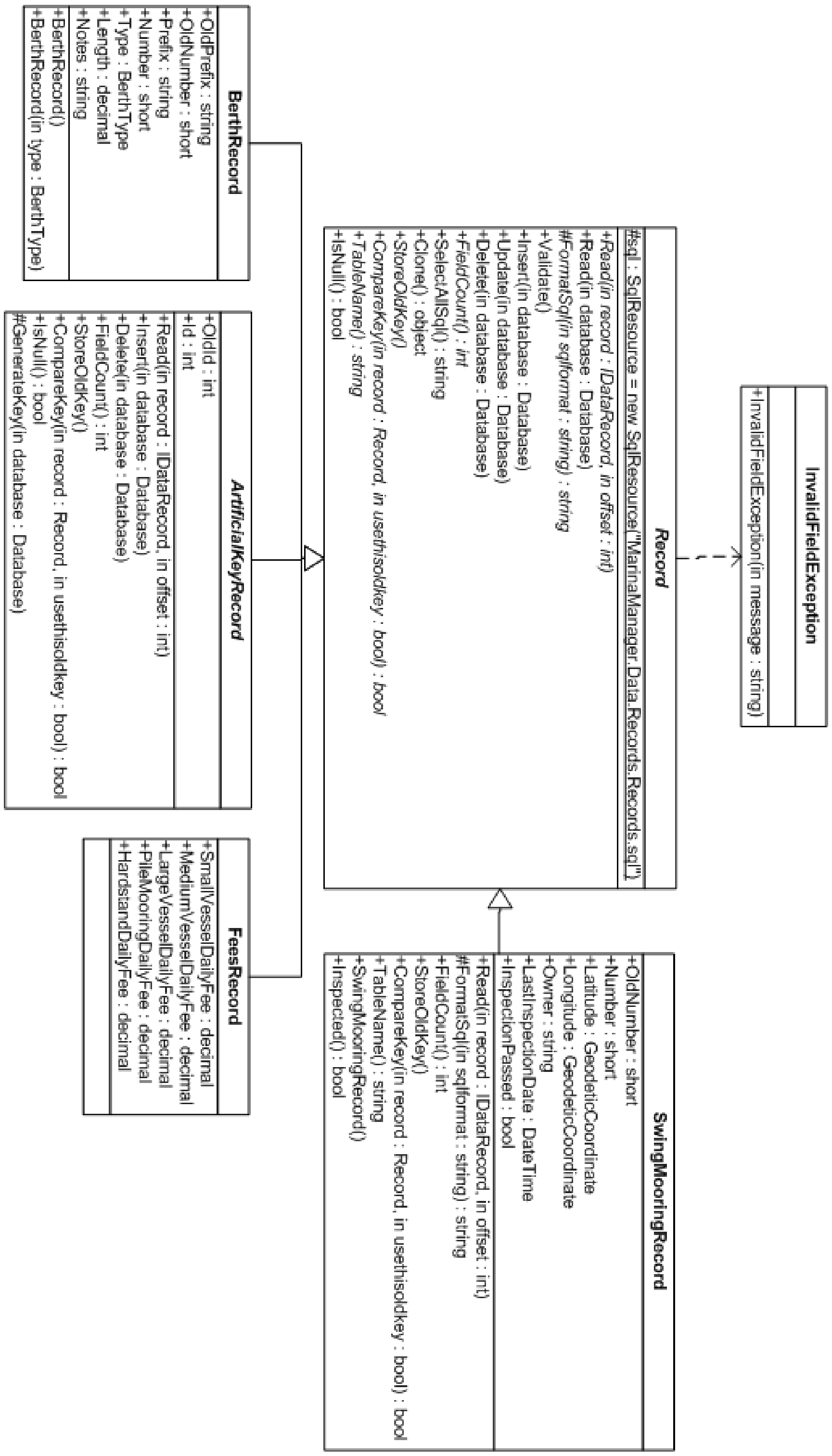

Setup Berths
Setup Pontoons
Setup Pole Moorings
Setup Hardstand Compound
Setup Storage Compound
Setup Fees
Marina Supervisor
Backup Data
Restore Backup
Save Changelog
Update from Changelog

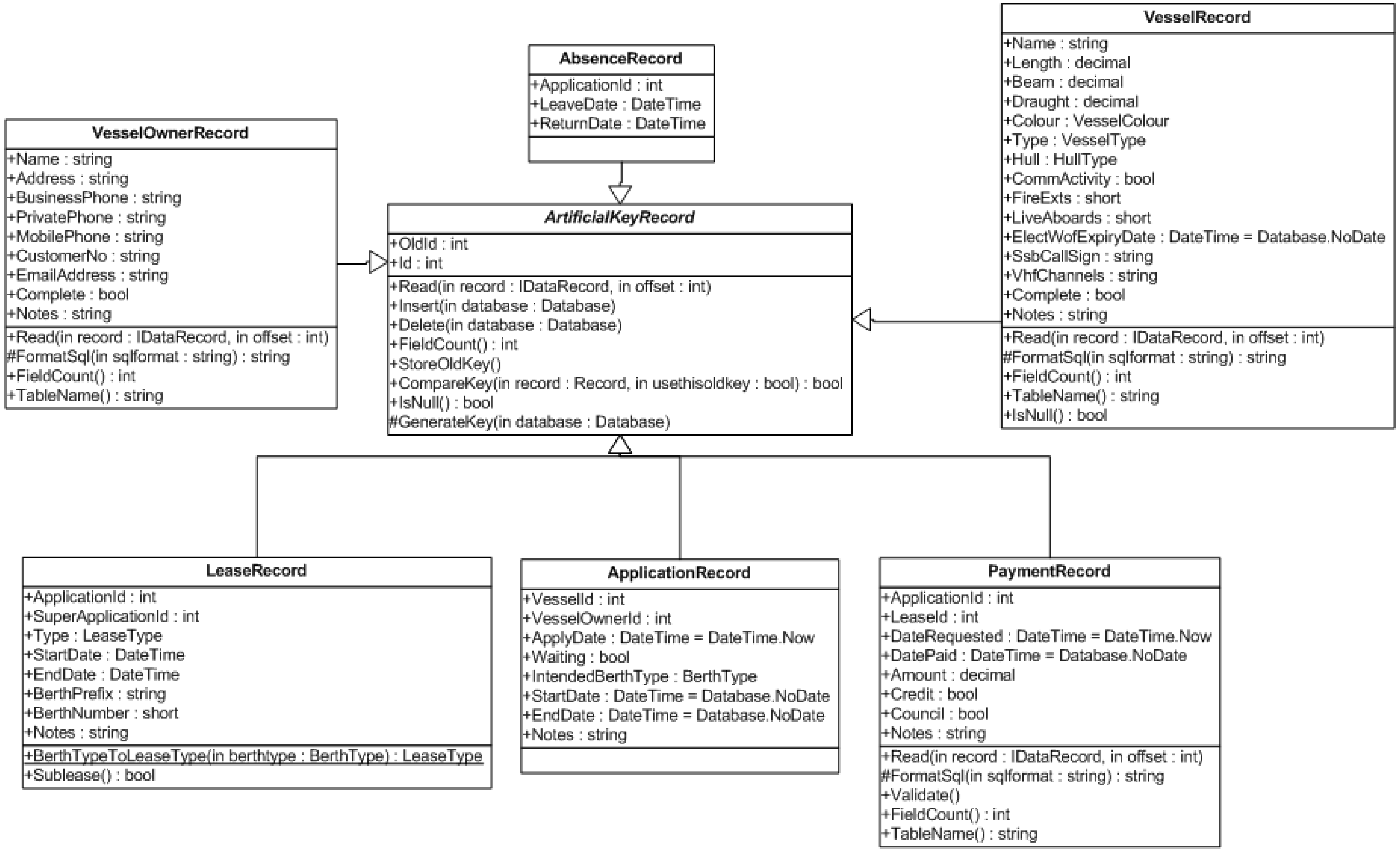
APPENDIX F: Class Designs

MarinaManager.Data

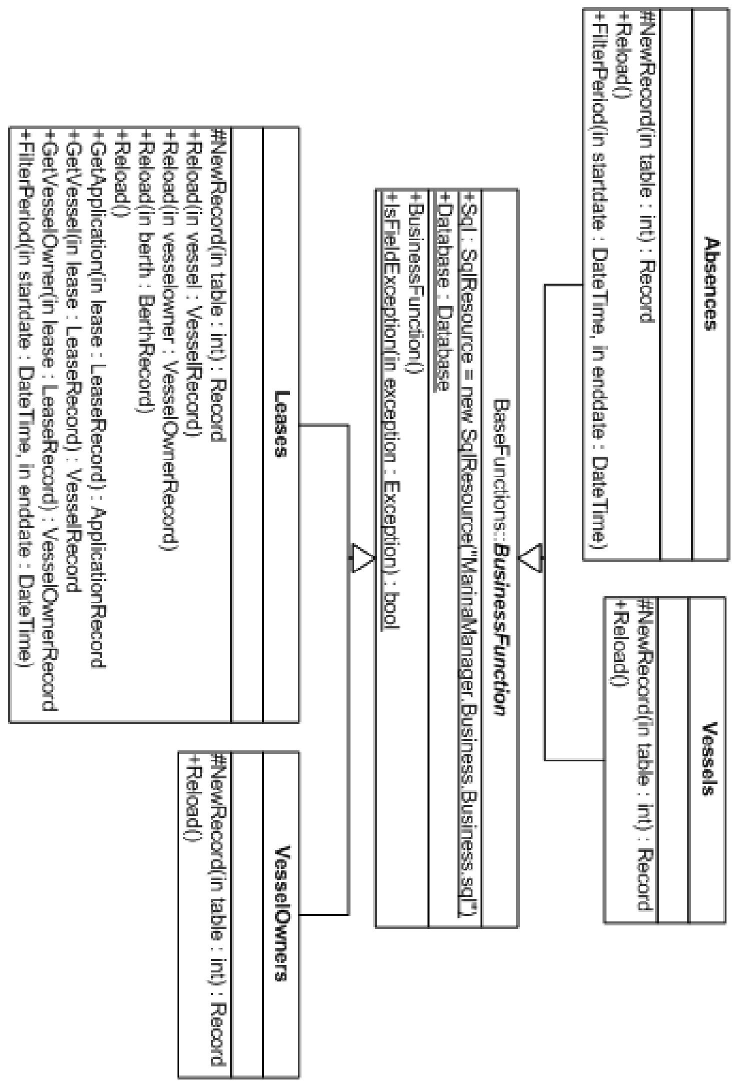

Function
#database : Database
-jointables : int = 1
-LoadRecord(inout record : Record, in reader : IDataReader, inout offset : int)
#NewRecord(in table : int) : Record
#Execute(in sql : string, in jointables : int)
#this(in index : int, in table : int) : Record
+this(in index : int) : Record
#IndexOf(in record : Record, in table : int) : int
+IndexOf(in record : Record) : int
#AddRecords(in records : params Record[])
Database
+NoDate : DateTime = new DateTime(1753, 1, 1)
-connection : SqlConnection
-log : SqlLog
+Name : string
+FileName : string
-operations : SqlResource = new SqlResource("MarinaManager.Data.Operations.sql")
-command : SqlCommand
+Database(in name : string, in filename : string)
+Close()
-DatabaseExists() : bool
-CreateObjects()
+ExecuteRead(in sql : string) : IDataReader
+ExecuteUpdate(in sql : string) : int
+ExecuteUnloggedUpdate(in sql : string) : int
+ExecuteScalarRead(in sql : string) : object
+Log() : SqlLog
+Backup(in filepath : string)
+Restore(in filepath : string)
+Recreate()
+FormatDate(in date : DateTime) : string
1
1
-operations
1
-log
1
SqlResource
#statementseparator : string = "\r\nGO"
#statementnamestart : string = "/*"
#statementnameend : string = "*/"
#itemsbyindex : ArrayList = new ArrayList()
#itemsbyname : Hashtable = new Hashtable()
#SqlResource()
+SqlResource(in resourcename : string)
#LoadSql(in reader : StreamReader) : string
#Fill(in sql : string)
-ExtractStatementName(inout sqlstatement : string) : string
+this(in statementname : string) : string
+this(in index : int) : string
+Count() : int
SqlLog
+SqlLog()
+Add(in sql : string)
+Execute(in database : Database) : bool
+Load(in path : string)
+Save(in path : string)

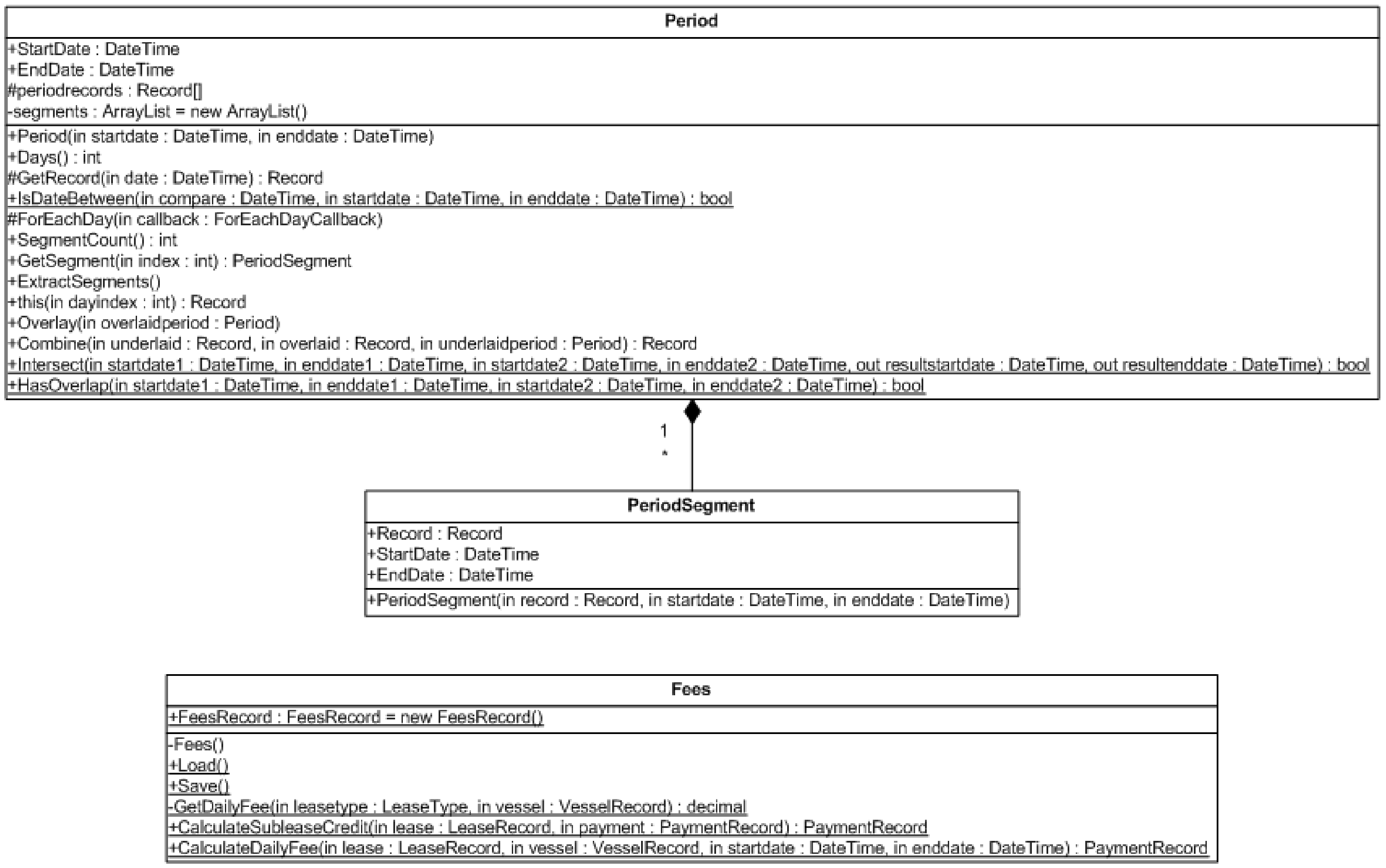
MarinaManager.Data.Records


BerthRecord
+OldPrefix : string
+OldNumber : short
+Prefix : string
+Number : short
+Type : BerthType
+Length : decimal
+Notes : string
+BerthRecord()
+BerthRecord(in type : BerthType)
ArtificialKeyRecord
+OldId : int
+Id : int
+Read(in record : IDataRecord, in offset : int)
+Insert(in database : Database)
+Delete(in database : Database)
+FieldCount() : int
+StoreOldKey()
+CompareKey(in record : Record, in usethisoldkey : bool) : bool
+IsNull() : bool
#GenerateKey(in database : Database)
FeesRecord
+SmallVesselDailyFee : decimal
+MediumVesselDailyFee : decimal
+LargeVesselDailyFee : decimal
+PileMooringDailyFee : decimal
+HardstandDailyFee : decimal
Record
#sql : SqlResource = new SqlResource("MarinaManager.Data.Records.Records.sql")
+Read(in record : IDataRecord, in offset : int)
+Read(in database : Database)
#FormatSql(in sqlformat : string) : string
+Validate()
+Insert(in database : Database)
+Update(in database : Database)
+Delete(in database : Database)
+FieldCount() : int
+SelectAllSql() : string
+Clone() : object
+StoreOldKey()
+CompareKey(in record : Record, in usethisoldkey : bool) : bool
+TableName() : string
+IsNull() : bool
InvalidFieldException
+InvalidFieldException(in message : string)
SwingMooringRecord
+OldNumber : short
+Number : short
+Latitude : GeodeticCoordinate
+Longitude : GeodeticCoordinate
+Owner : string
+LastInspectionDate : DateTime
+InspectionPassed : bool
+Read(in record : IDataRecord, in offset : int)
#FormatSql(in sqlformat : string) : string
+FieldCount() : int
+StoreOldKey()
+CompareKey(in record : Record, in usethisoldkey : bool) : bool
+TableName() : string
+SwingMooringRecord()
+Inspected() : bool

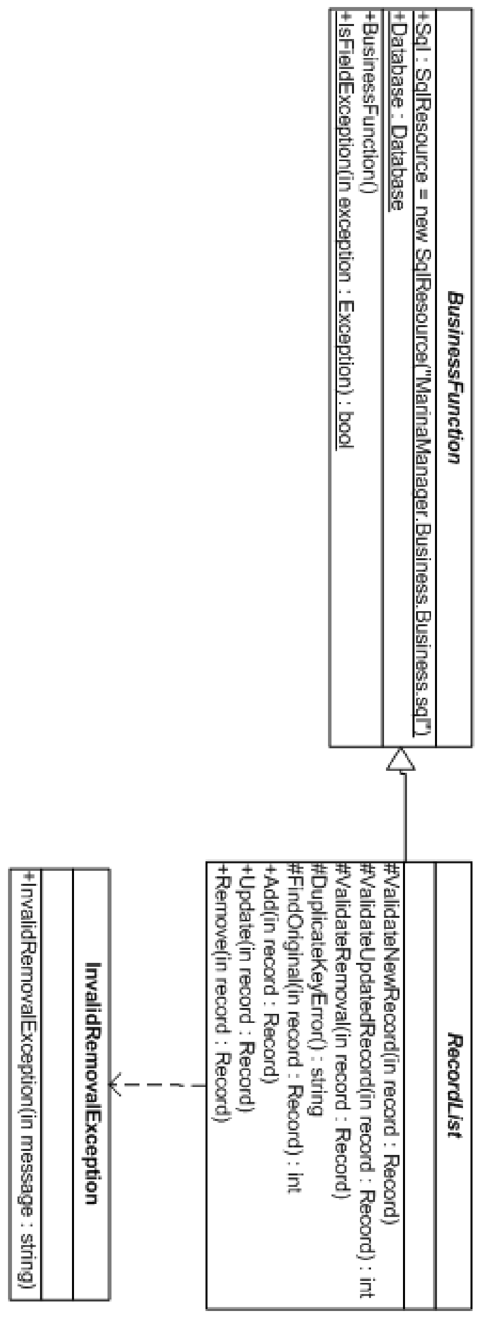

VesselOwnerRecord
+Name : string
+Address : string
+BusinessPhone : string
+PrivatePhone : string
+MobilePhone : string
+CustomerNo : string
+EmailAddress : string
+Complete : bool
+Notes : string
+Read(in record : IDataRecord, in offset : int)
#FormatSql(in sqlformat : string) : string
+FieldCount() : int
+TableName() : string
AbsenceRecord
+ApplicationId : int
+LeaveDate : DateTime
+ReturnDate : DateTime
ArtificialKeyRecord
+OldId : int
+Id : int
+Read(in record : IDataRecord, in offset : int)
+Insert(in database : Database)
+Delete(in database : Database)
+FieldCount() : int
+StoreOldKey()
+CompareKey(in record : Record, in usethisoldkey : bool) : bool
+IsNull() : bool
#GenerateKey(in database : Database)
VesselRecord
+Name : string
+Length : decimal
+Beam : decimal
+Draught : decimal
+Colour : VesselColour
+Type : VesselType
+Hull : HullType
+CommActivity : bool
+FireExts : short
+LiveAboards : short
+ElectWofExpiryDate : DateTime = Database.NoDate
+SsbCallSign : string
+VhfChannels : string
+Complete : bool
+Notes : string
+Read(in record : IDataRecord, in offset : int)
#FormatSql(in sqlformat : string) : string
+FieldCount() : int
+TableName() : string
+IsNull() : bool
LeaseRecord
+ApplicationId : int
+SuperApplicationId : int
+Type : LeaseType
+StartDate : DateTime
+EndDate : DateTime
+BerthPrefix : string
+BerthNumber : short
+Notes : string
+BerthTypeToLeaseType(in berthtype : BerthType) : LeaseType
+Sublease() : bool
ApplicationRecord
+VesselId : int
+VesselOwnerId : int
+ApplyDate : DateTime = DateTime.Now
+Waiting : bool
+IntendedBerthType : BerthType
+StartDate : DateTime = Database.NoDate
+EndDate : DateTime = Database.NoDate
+Notes : string
PaymentRecord
+ApplicationId : int
+LeaseId : int
+DateRequested : DateTime = DateTime.Now
+DatePaid : DateTime = Database.NoDate
+Amount : decimal
+Credit : bool
+Council : bool
+Notes : string
+Read(in record : IDataRecord, in offset : int)
#FormatSql(in sqlformat : string) : string
+Validate()
+FieldCount() : int
+TableName() : string

MarinaManager.Business

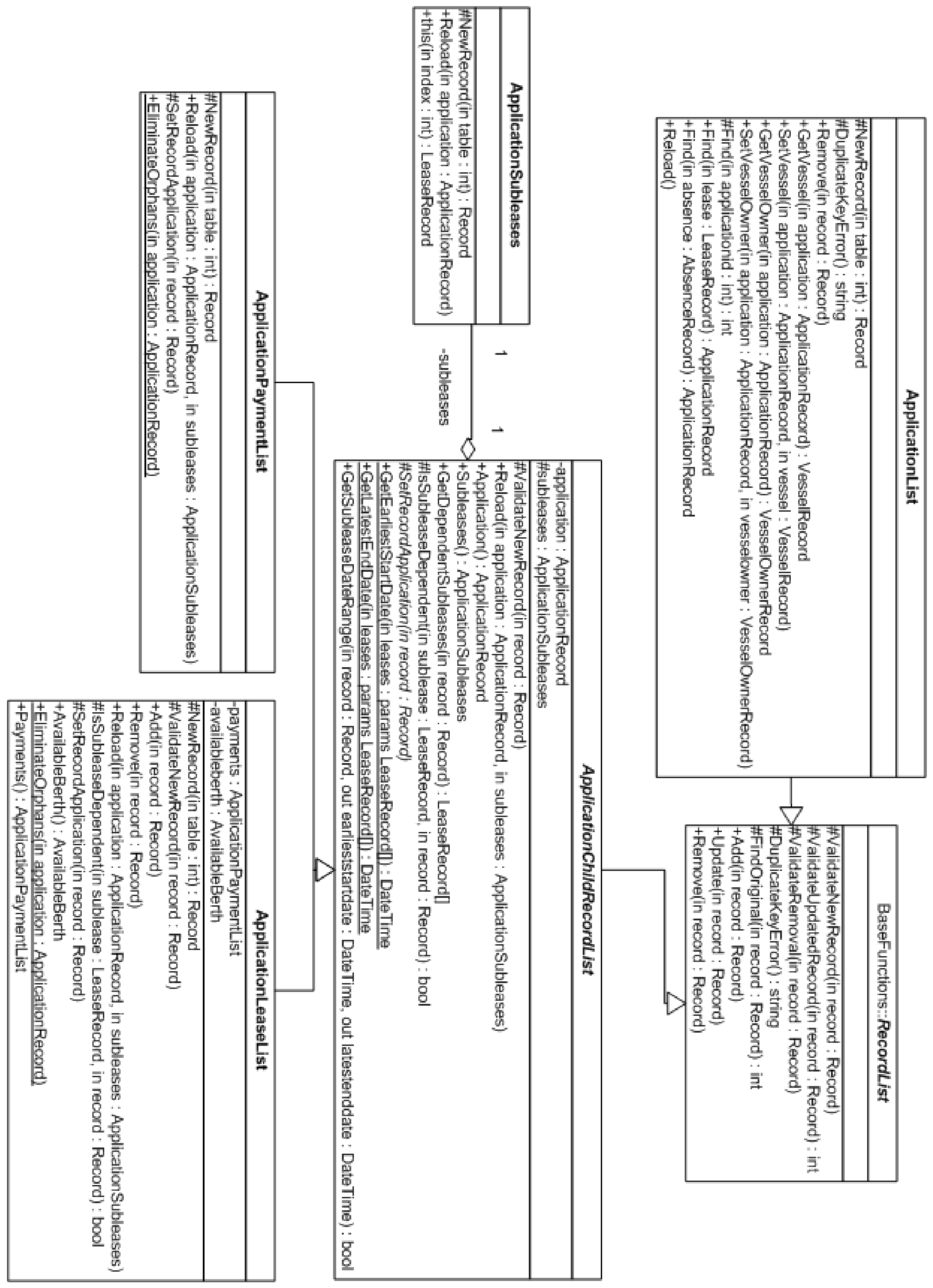

Absences
#NewRecord(in table : int) : Record
+Reload()
+FilterPeriod(in startdate : DateTime, in enddate : DateTime)
Vessels
#NewRecord(in table : int) : Record
+Reload()
BaseFunctions::BusinessFunction
+Sql : SqlResource = new SqlResource("MarinaManager.Business.Business.sql")
+Database : Database
+BusinessFunction()
+IsFieldException(in exception : Exception) : bool
Leases
#NewRecord(in table : int) : Record
+Reload(in vessel : VesselRecord)
+Reload(in vesselowner : VesselOwnerRecord)
+Reload(in berth : BerthRecord)
+Reload()
+GetApplication(in lease : LeaseRecord) : ApplicationRecord
+GetVessel(in lease : LeaseRecord) : VesselRecord
+GetVesselOwner(in lease : LeaseRecord) : VesselOwnerRecord
+FilterPeriod(in startdate : DateTime, in enddate : DateTime)
VesselOwners
#NewRecord(in table : int) : Record
+Reload()


Period
+StartDate : DateTime
+EndDate : DateTime
#periodrecords : Record[]
-segments : ArrayList = new ArrayList()
+Period(in startdate : DateTime, in enddate : DateTime)
+Days() : int
#GetRecord(in date : DateTime) : Record
+IsDateBetween(in compare : DateTime, in startdate : DateTime, in enddate : DateTime) : bool
#ForEachDay(in callback : ForEachDayCallback)
+SegmentCount() : int
+GetSegment(in index : int) : PeriodSegment
+ExtractSegments()
+this(in dayindex : int) : Record
+Overlay(in overlaidperiod : Period)
+Combine(in underlaid : Record, in overlaid : Record, in underlaidperiod : Period) : Record
+Intersect(in startdate1 : DateTime, in enddate1 : DateTime, in startdate2 : DateTime, in enddate2 : DateTime, out resultstartdate : DateTime, out resultenddate : DateTime) : bool
+HasOverlap(in startdate1 : DateTime, in enddate1 : DateTime, in startdate2 : DateTime, in enddate2 : DateTime) : bool
1
*
PeriodSegment
+Record : Record
+StartDate : DateTime
+EndDate : DateTime
+PeriodSegment(in record : Record, in startdate : DateTime, in enddate : DateTime)
Fees
+FeesRecord : FeesRecord = new FeesRecord()
-Fees()
+Load()
+Save()
-GetDailyFee(in leasetype : LeaseType, in vessel : VesselRecord) : decimal
+CalculateSubleaseCredit(in lease : LeaseRecord, in payment : PaymentRecord) : PaymentRecord
+CalculateDailyFee(in lease : LeaseRecord, in vessel : VesselRecord, in startdate : DateTime, in enddate : DateTime) : PaymentRecord

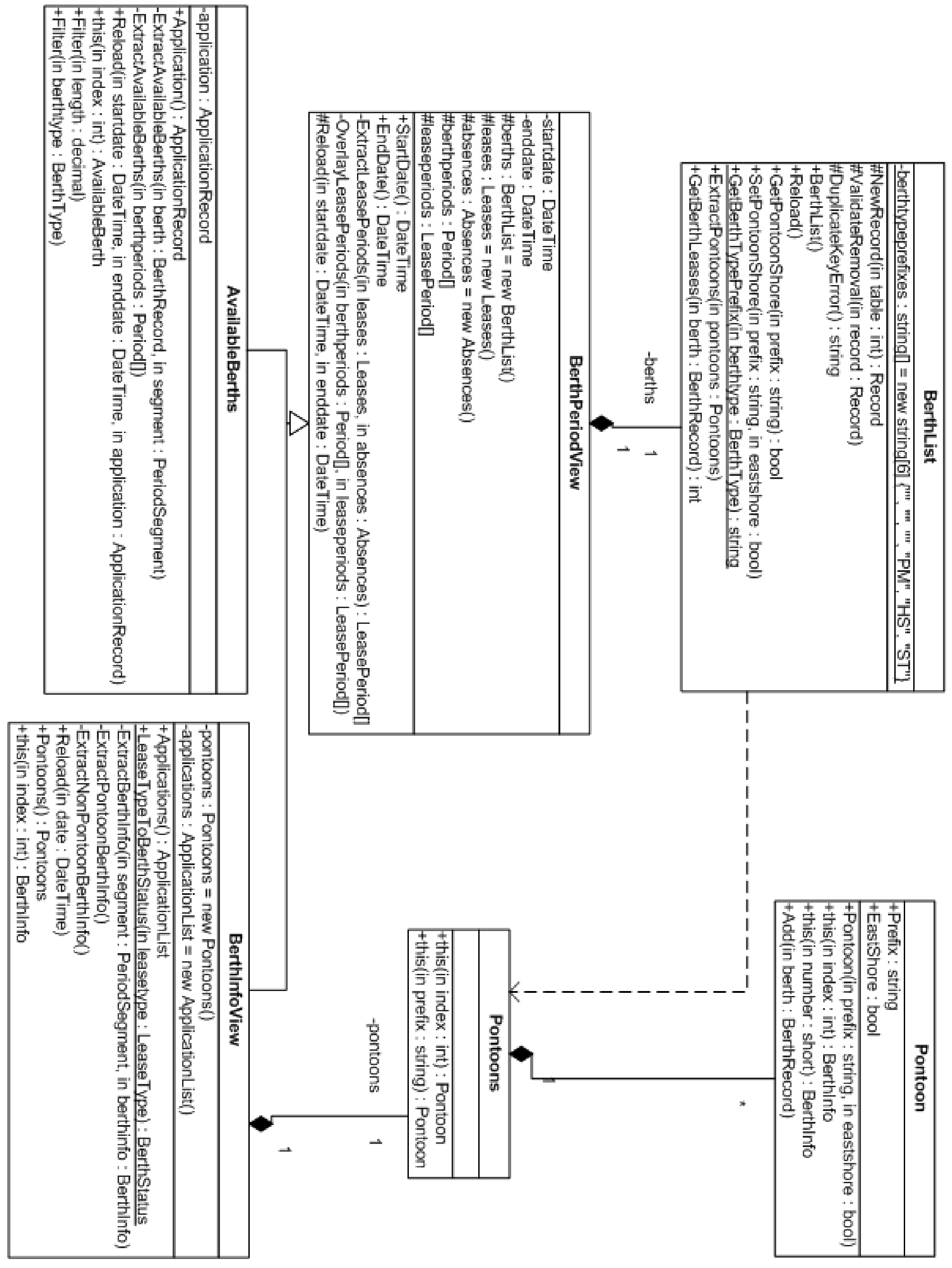
MarinaManager.Business.BaseFunctions

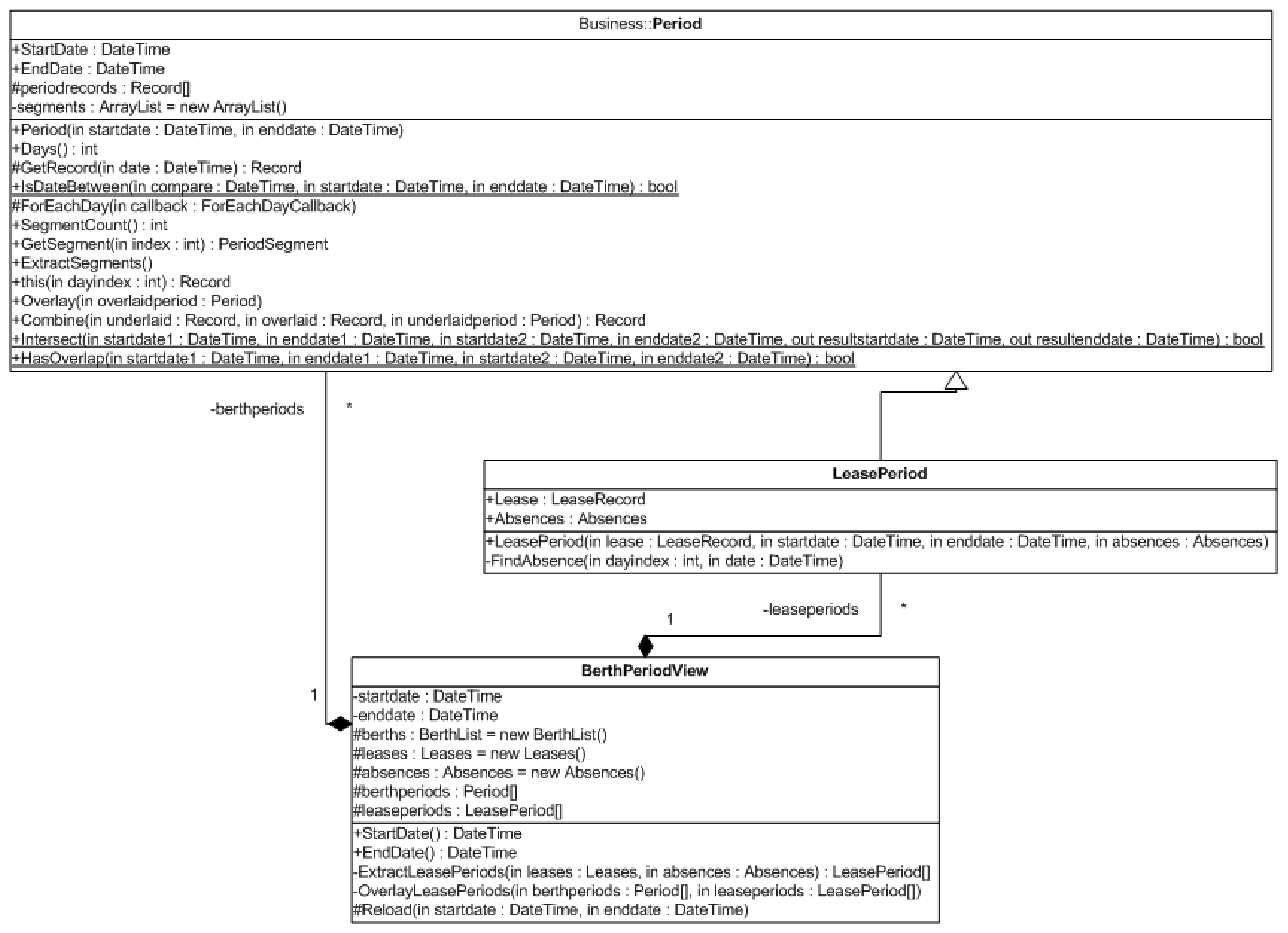

BusinessFunction
+Sql : SqlResource = new SqlResource("MarinaManager.Business.Business.sql")
+Database : Database
+BusinessFunction()
+IsFieldException(in exception : Exception) : bool
RecordList
#ValidateNewRecord(in record : Record)
#ValidateUpdatedRecord(in record : Record) : int
#ValidateRemoval(in record : Record)
#DuplicateKeyError() : string
#FindOriginal(in record : Record) : int
+Add(in record : Record)
+Update(in record : Record)
+Remove(in record : Record)
InvalidRemovalException
+InvalidRemovalException(in message : string)

MarinaManager.Business.Applications

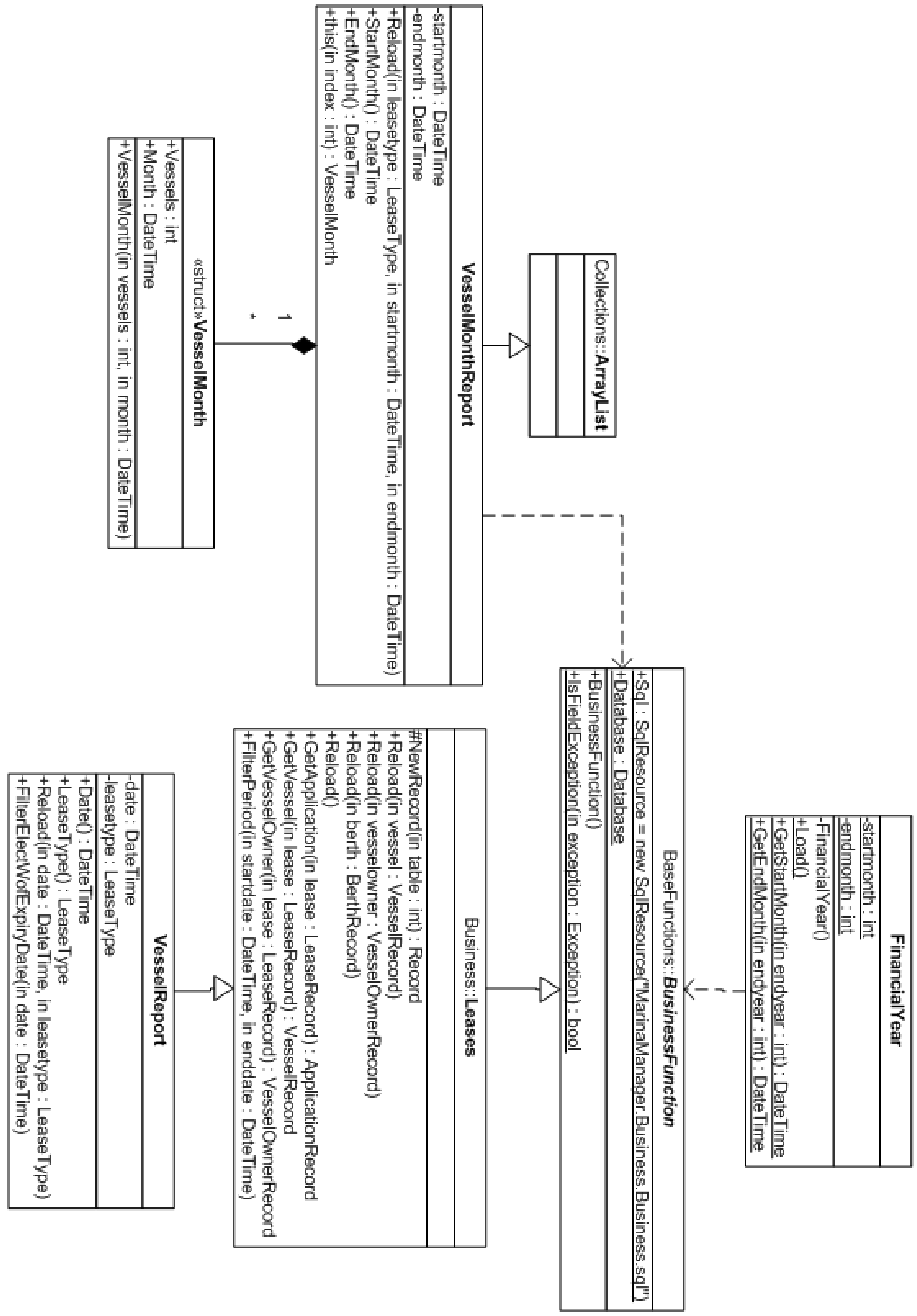

ApplicationList
#NewRecord(in table : int) : Record
#DuplicateKeyError() : string
+Remove(in record : Record)
+GetVessel(in application : ApplicationRecord) : VesselRecord
+SetVessel(in application : ApplicationRecord, in vessel : VesselRecord)
+GetVesselOwner(in application : ApplicationRecord) : VesselOwnerRecord
+SetVesselOwner(in application : ApplicationRecord, in vesselowner : VesselOwnerRecord)
#Find(in applicationid : int) : int
+Find(in lease : LeaseRecord) : ApplicationRecord
+Find(in absence : AbsenceRecord) : ApplicationRecord
+Reload()
BaseFunctions::RecordList
#ValidateNewRecord(in record : Record)
#ValidateUpdatedRecord(in record : Record) : int
#ValidateRemoval(in record : Record)
#DuplicateKeyError() : string
#FindOriginal(in record : Record) : int
+Add(in record : Record)
+Update(in record : Record)
+Remove(in record : Record)
ApplicationSubleases
#NewRecord(in table : int) : Record
+Reload(in application : ApplicationRecord)
+this(in index : int) : LeaseRecord
1
-subleases
1
ApplicationChildRecordList
-application : ApplicationRecord
#subleases : ApplicationSubleases
#ValidateNewRecord(in record : Record)
+Reload(in application : ApplicationRecord, in subleases : ApplicationSubleases)
+Application() : ApplicationRecord
+Subleases() : ApplicationSubleases
+GetDependentSubleases(in record : Record) : LeaseRecord[]
#IsSubleaseDependent(in sublease : LeaseRecord, in record : Record) : bool
#SetRecordApplication(in record : Record)
+GetEarliestStartDate(in leases : params LeaseRecord[]) : DateTime
+GetLatestEndDate(in leases : params LeaseRecord[]) : DateTime
+GetSubleaseDateRange(in record : Record, out earlieststartdate : DateTime, out latestenddate : DateTime) : bool
ApplicationPaymentList
#NewRecord(in table : int) : Record
+Reload(in application : ApplicationRecord, in subleases : ApplicationSubleases)
#SetRecordApplication(in record : Record)
+EliminateOrphans(in application : ApplicationRecord)
ApplicationLeaseList
-payments : ApplicationPaymentList
-availableberth : AvailableBerth
#NewRecord(in table : int) : Record
#ValidateNewRecord(in record : Record)
+Add(in record : Record)
+Remove(in record : Record)
+Reload(in application : ApplicationRecord, in subleases : ApplicationSubleases)
#IsSubleaseDependent(in sublease : LeaseRecord, in record : Record) : bool
#SetRecordApplication(in record : Record)
+AvailableBerth() : AvailableBerth
+EliminateOrphans(in application : ApplicationRecord)
+Payments() : ApplicationPaymentList

MarinaManager.Business.Berths

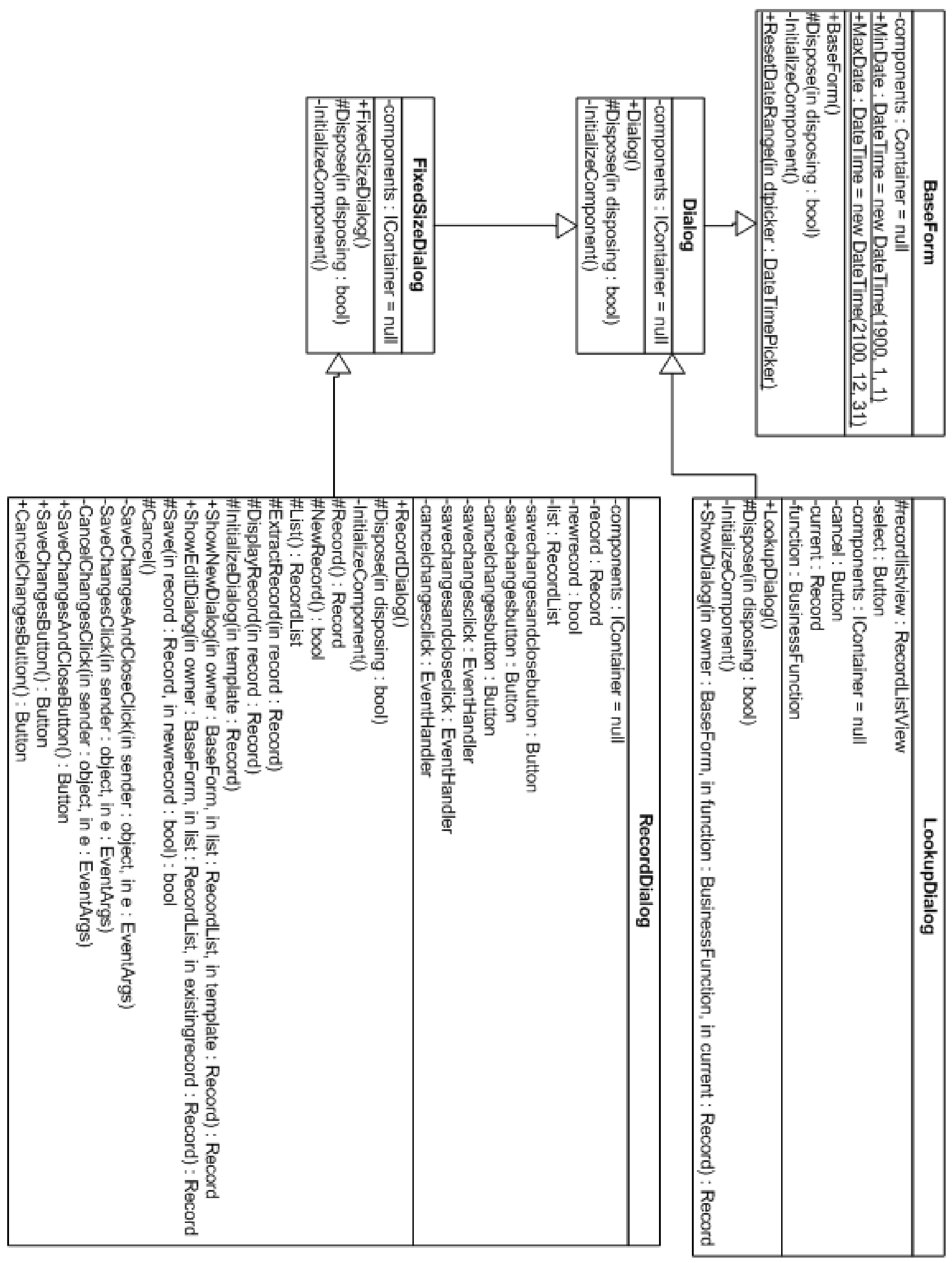

BerthList
-berthtypeprefixes : string[] = new string[6] {"", "", "", "PM", "HS", "ST"}
#NewRecord(in table : int) : Record
#ValidateRemoval(in record : Record)
#DuplicateKeyError() : string
+BerthList()
+Reload()
+GetPontoonShore(in prefix : string) : bool
+SetPontoonShore(in prefix : string, in eastshore : bool)
+GetBerthTypePrefix(in berthtype : BerthType) : string
+ExtractPontoons(in pontoons : Pontoons)
+GetBerthLeases(in berth : BerthRecord) : int
-berths
1
1
BerthPeriodView
-startdate : DateTime
-enddate : DateTime
#berths : BerthList = new BerthList()
#leases : Leases = new Leases()
#absences : Absences = new Absences()
#berthperiods : Period[]
#leaseperiods : LeasePeriod[]
+StartDate() : DateTime
+EndDate() : DateTime
-ExtractLeasePeriods(in leases : Leases, in absences : Absences) : LeasePeriod[]
-OverlayLeasePeriods(in berthperiods : Period[], in leaseperiods : LeasePeriod[])
#Reload(in startdate : DateTime, in enddate : DateTime)
AvailableBerths
-application : ApplicationRecord
+Application() : ApplicationRecord
-ExtractAvailableBerths(in berth : BerthRecord, in segment : PeriodSegment)
-ExtractAvailableBerths(in berthperiods : Period[])
+Reload(in startdate : DateTime, in enddate : DateTime, in application : ApplicationRecord)
+this(in index : int) : AvailableBerth
+Filter(in length : decimal)
+Filter(in berthtype : BerthType)
BerthInfoView
-pontoons : Pontoons = new Pontoons()
-applications : ApplicationList = new ApplicationList()
+Applications() : ApplicationList
+LeaseTypeToBerthStatus(in leasetype : LeaseType) : BerthStatus
-ExtractBerthInfo(in segment : PeriodSegment, in berthinfo : BerthInfo)
-ExtractPontoonBerthInfo()
-ExtractNonPontoonBerthInfo()
+Reload(in date : DateTime)
+Pontoons() : Pontoons
+this(in index : int) : BerthInfo
-pontoons
1
1
Pontoons
+this(in index : int) : Pontoon
+this(in prefix : string) : Pontoon
1
*
Pontoon
+Prefix : string
+EastShore : bool
+Pontoon(in prefix : string, in eastshore : bool)
+this(in index : int) : BerthInfo
+this(in number : short) : BerthInfo
+Add(in berth : BerthRecord)


Business::**Period**
+StartDate : DateTime
+EndDate : DateTime
#periodrecords : Record[]
-segments : ArrayList = new ArrayList()
+Period(in startdate : DateTime, in enddate : DateTime)
+Days() : int
#GetRecord(in date : DateTime) : Record
+IsDateBetween(in compare : DateTime, in startdate : DateTime, in enddate : DateTime) : bool
#ForEachDay(in callback : ForEachDayCallback)
+SegmentCount() : int
+GetSegment(in index : int) : PeriodSegment
+ExtractSegments()
+this(in dayindex : int) : Record
+Overlay(in overlaidperiod : Period)
+Combine(in underlaid : Record, in overlaid : Record, in underlaidperiod : Period) : Record
+Intersect(in startdate1 : DateTime, in enddate1 : DateTime, in startdate2 : DateTime, in enddate2 : DateTime, out resultstartdate : DateTime, out resultenddate : DateTime) : bool
+HasOverlap(in startdate1 : DateTime, in enddate1 : DateTime, in startdate2 : DateTime, in enddate2 : DateTime) : bool
-berthperiods
*
LeasePeriod
+Lease : LeaseRecord
+Absences : Absences
+LeasePeriod(in lease : LeaseRecord, in startdate : DateTime, in enddate : DateTime, in absences : Absences)
-FindAbsence(in dayindex : int, in date : DateTime)
1
-leaseperiods
*
BerthPeriodView
1
-startdate : DateTime
-enddate : DateTime
#berths : BerthList = new BerthList()
#leases : Leases = new Leases()
#absences : Absences = new Absences()
#berthperiods : Period[]
#leaseperiods : LeasePeriod[]
+StartDate() : DateTime
+EndDate() : DateTime
-ExtractLeasePeriods(in leases : Leases, in absences : Absences) : LeasePeriod[]
-OverlayLeasePeriods(in berthperiods : Period[], in leaseperiods : LeasePeriod[])
#Reload(in startdate : DateTime, in enddate : DateTime)

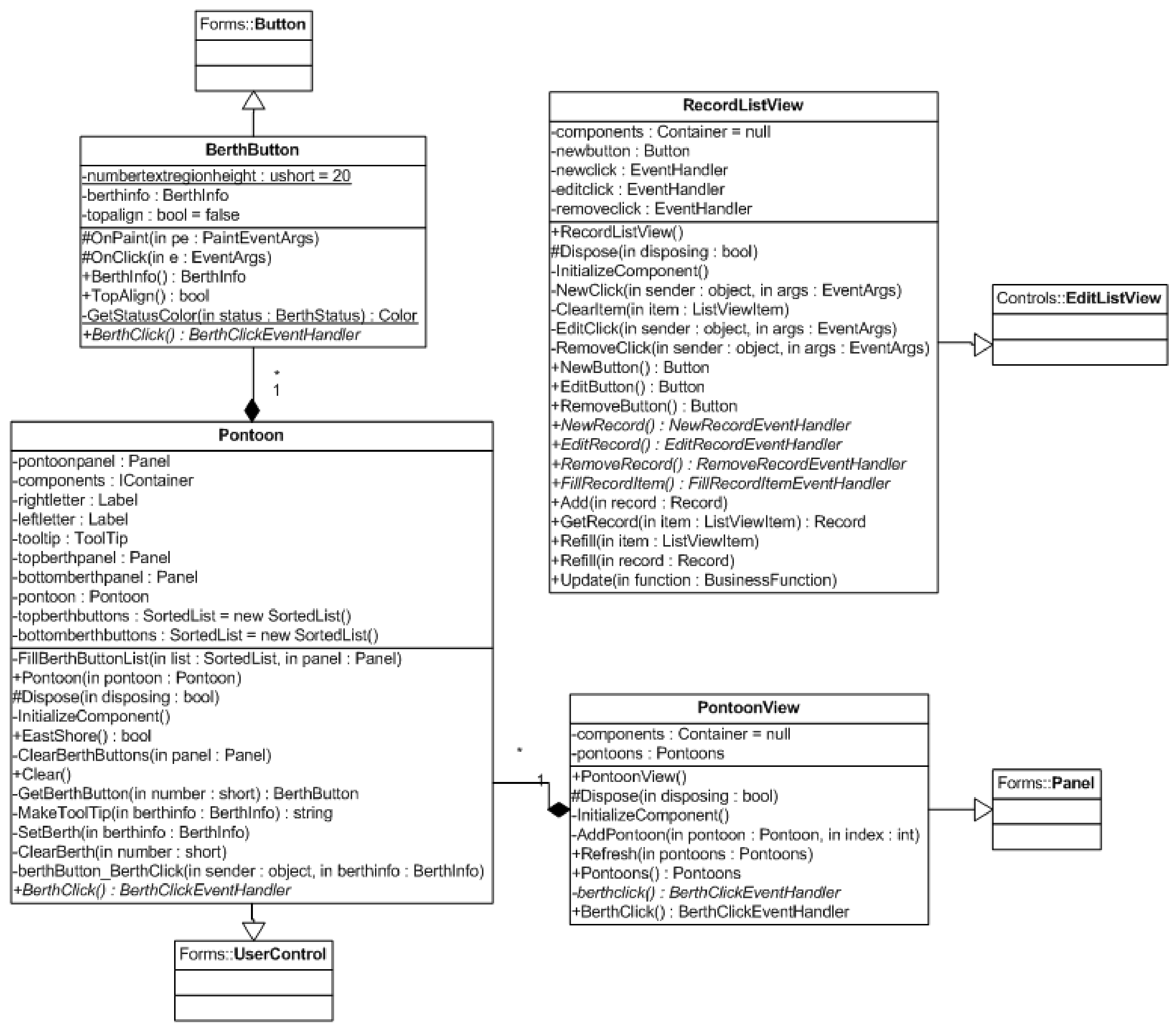
MarinaManager.Business.Reporting


VesselMonthReport
-startmonth : DateTime
-endmonth : DateTime
+Reload(in leasetype : LeaseType, in startmonth : DateTime, in endmonth : DateTime)
+StartMonth() : DateTime
+EndMonth() : DateTime
+this(in index : int) : VesselMonth
Collections::ArrayList
«struct»VesselMonth
+Vessels : int
+Month : DateTime
+VesselMonth(in vessels : int, in month : DateTime)
1
*
BaseFunctions::BusinessFunction
+Sql : SqlResource = new SqlResource("MarinaManager.Business.Business.sql")
+Database : Database
+BusinessFunction()
+IsFieldException(in exception : Exception) : bool
FinancialYear
-startmonth : int
-endmonth : int
-FinancialYear()
+Load()
+GetStartMonth(in endyear : int) : DateTime
+GetEndMonth(in endyear : int) : DateTime
Business::Leases
#NewRecord(in table : int) : Record
+Reload(in vessel : VesselRecord)
+Reload(in vesselowner : VesselOwnerRecord)
+Reload(in berth : BerthRecord)
+Reload()
+GetApplication(in lease : LeaseRecord) : ApplicationRecord
+GetVessel(in lease : LeaseRecord) : VesselRecord
+GetVesselOwner(in lease : LeaseRecord) : VesselOwnerRecord
+FilterPeriod(in startdate : DateTime, in enddate : DateTime)
VesselReport
-date : DateTime
-leasetype : LeaseType
+Date() : DateTime
+LeaseType() : LeaseType
+Reload(in date : DateTime, in leasetype : LeaseType)
+FilterElectWofExpiryDate(in date : DateTime)

MarinaManager.UI.BaseForms

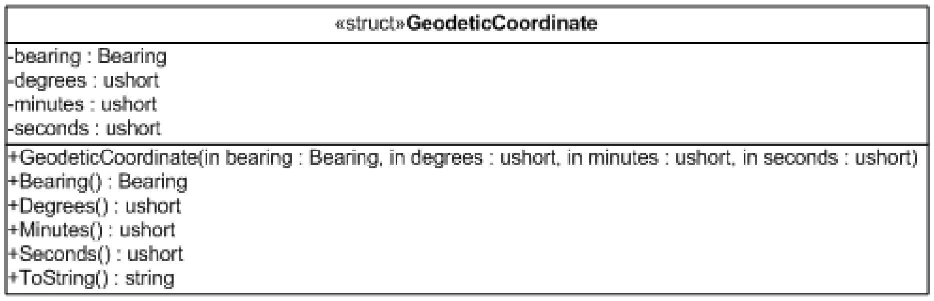

BaseForm
-components : Container = null
+MinDate : DateTime = new DateTime(1900, 1, 1)
+MaxDate : DateTime = new DateTime(2100, 12, 31)
+BaseForm()
#Dispose(in disposing : bool)
-InitializeComponent()
+ResetDateRange(in dtpicker : DateTimePicker)
Dialog
-components : IContainer = null
+Dialog()
#Dispose(in disposing : bool)
-InitializeComponent()
FixedSizeDialog
-components : IContainer = null
+FixedSizeDialog()
#Dispose(in disposing : bool)
-InitializeComponent()
LookupDialog
#recordlistview : RecordListView
-select : Button
-components : IContainer = null
-cancel : Button
-current : Record
-function : BusinessFunction
+LookupDialog()
#Dispose(in disposing : bool)
-InitializeComponent()
+ShowDialog(in owner : BaseForm, in function : BusinessFunction, in current : Record) : Record
RecordDialog
-components : IContainer = null
-record : Record
-newrecord : bool
-list : RecordList
-savechangesandclosebutton : Button
-savechangesbutton : Button
-cancelchangesbutton : Button
-savechangesclick : EventHandler
-savechangesandcloseclick : EventHandler
-cancelchangesclick : EventHandler
+RecordDialog()
#Dispose(in disposing : bool)
-InitializeComponent()
#Record() : Record
#NewRecord() : bool
#List() : RecordList
#ExtractRecord(in record : Record)
#DisplayRecord(in record : Record)
#InitializeDialog(in template : Record)
+ShowNewDialog(in owner : BaseForm, in list : RecordList, in template : Record) : Record
+ShowEditDialog(in owner : BaseForm, in list : RecordList, in existingrecord : Record) : Record
#Save(in record : Record, in newrecord : bool) : bool
#Cancel()
-SaveChangesAndCloseClick(in sender : object, in e : EventArgs)
-SaveChangesClick(in sender : object, in e : EventArgs)
-CancelChangesClick(in sender : object, in e : EventArgs)
+SaveChangesAndCloseButton() : Button
+SaveChangesButton() : Button
+CancelChangesButton() : Button

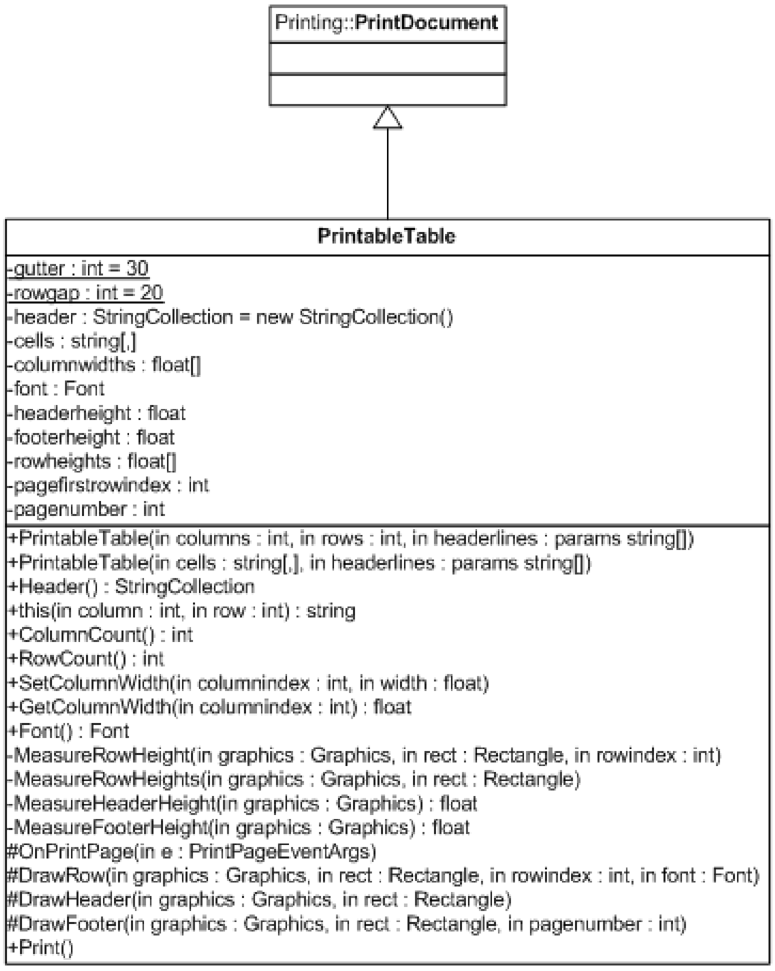
MarinaManager.UI.Controls



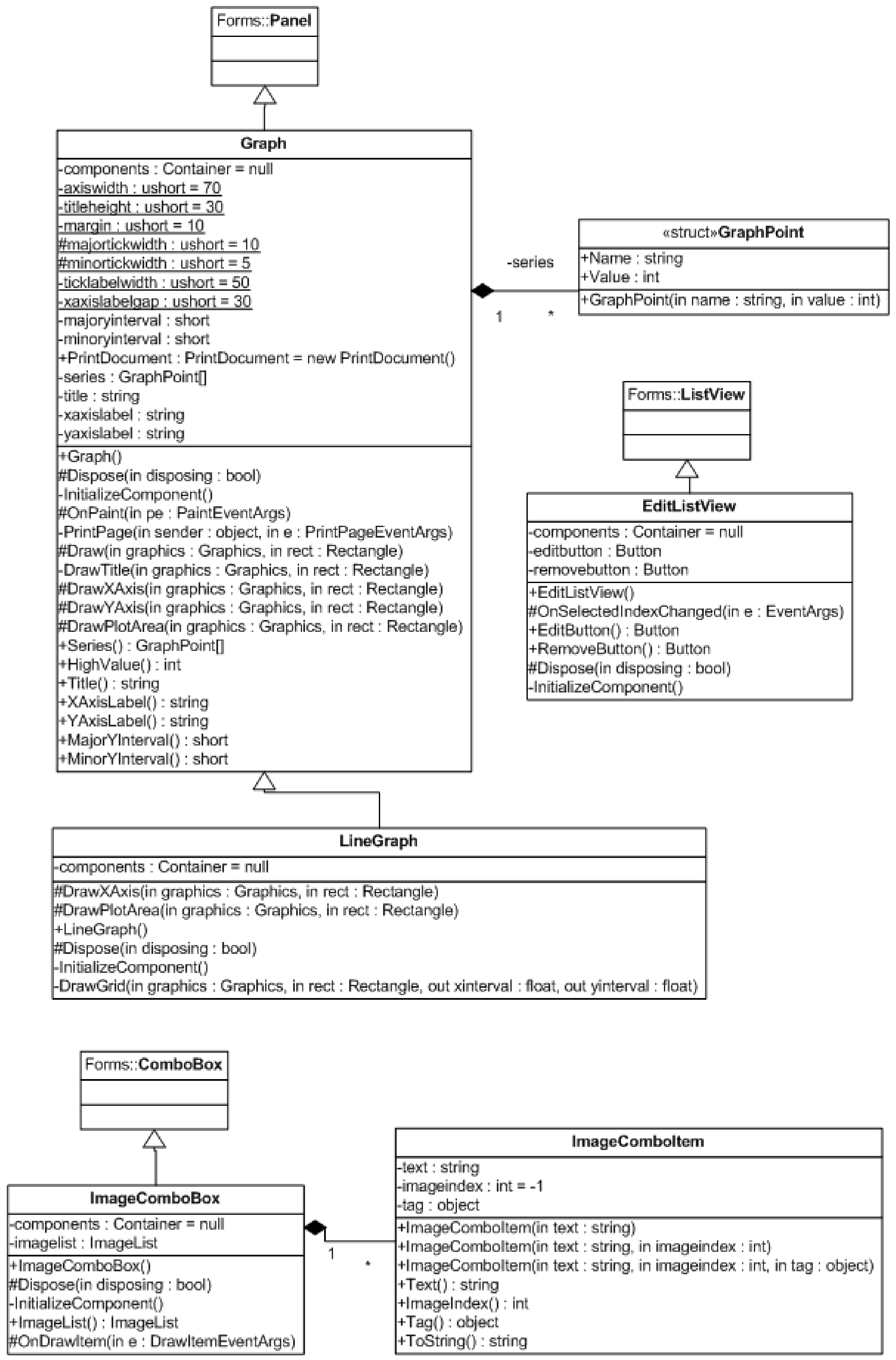
Forms::Button
BerthButton
-numbertextregionheight : ushort = 20
-berthinfo : BerthInfo
-topalign : bool = false
#OnPaint(in pe : PaintEventArgs)
#OnClick(in e : EventArgs)
+BerthInfo() : BerthInfo
+TopAlign() : bool
-GetStatusColor(in status : BerthStatus) : Color
+BerthClick() : BerthClickEventHandler
*
1
Pontoon
-pontoonpanel : Panel
-components : IContainer
-rightletter : Label
-leftletter : Label
-tooltip : ToolTip
-topberthpanel : Panel
-bottomberthpanel : Panel
-pontoon : Pontoon
-topberthbuttons : SortedList = new SortedList()
-bottomberthbuttons : SortedList = new SortedList()
-FillBerthButtonList(in list : SortedList, in panel : Panel)
+Pontoon(in pontoon : Pontoon)
#Dispose(in disposing : bool)
-InitializeComponent()
+EastShore() : bool
-ClearBerthButtons(in panel : Panel)
+Clear()
-GetBerthButton(in number : short) : BerthButton
-MakeToolTip(in berthinfo : BerthInfo) : string
-SetBerth(in berthinfo : BerthInfo)
-ClearBerth(in number : short)
-berthButton_BerthClick(in sender : object, in berthinfo : BerthInfo)
+BerthClick() : BerthClickEventHandler
Forms::UserControl
*
1
PontoonView
-components : Container = null
-pontoons : Pontoons
+PontoonView()
#Dispose(in disposing : bool)
-InitializeComponent()
-AddPontoon(in pontoon : Pontoon, in index : int)
+Refresh(in pontoons : Pontoons)
+Pontoons() : Pontoons
-berthclick() : BerthClickEventHandler
+BerthClick() : BerthClickEventHandler
Forms::Panel
RecordListView
-components : Container = null
-newbutton : Button
-newclick : EventHandler
-editclick : EventHandler
-removeclick : EventHandler
+RecordListView()
#Dispose(in disposing : bool)
-InitializeComponent()
-NewClick(in sender : object, in args : EventArgs)
-ClearItem(in item : ListViewItem)
-EditClick(in sender : object, in args : EventArgs)
-RemoveClick(in sender : object, in args : EventArgs)
+NewButton() : Button
+EditButton() : Button
+RemoveButton() : Button
+NewRecord() : NewRecordEventHandler
+EditRecord() : EditRecordEventHandler
+RemoveRecord() : RemoveRecordEventHandler
+FillRecordItem() : FillRecordItemEventHandler
+Add(in record : Record)
+GetRecord(in item : ListViewItem) : Record
+Refill(in item : ListViewItem)
+Refill(in record : Record)
+Update(in function : BusinessFunction)
Controls::EditListView

MyLibrary


«struct»GeodeticCoordinate
-bearing : Bearing
-degrees : ushort
-minutes : ushort
-seconds : ushort
+GeodeticCoordinate(in bearing : Bearing, in degrees : ushort, in minutes : ushort, in seconds : ushort)
+Bearing() : Bearing
+Degrees() : ushort
+Minutes() : ushort
+Seconds() : ushort
+ToString() : string


Printing::PrintDocument
PrintableTable
-gutter : int = 30
-rowgap : int = 20
-header : StringCollection = new StringCollection()
-cells : string[,]
-columnwidths : float[]
-font : Font
-headerheight : float
-footerheight : float
-rowheights : float[]
-pagefirstrowindex : int
-pagenumber : int
+PrintableTable(in columns : int, in rows : int, in headerlines : params string[])
+PrintableTable(in cells : string[,], in headerlines : params string[])
+Header() : StringCollection
+this(in column : int, in row : int) : string
+ColumnCount() : int
+RowCount() : int
+SetColumnWidth(in columnindex : int, in width : float)
+GetColumnWidth(in columnindex : int) : float
+Font() : Font
-MeasureRowHeight(in graphics : Graphics, in rect : Rectangle, in rowindex : int)
-MeasureRowHeights(in graphics : Graphics, in rect : Rectangle)
-MeasureHeaderHeight(in graphics : Graphics) : float
-MeasureFooterHeight(in graphics : Graphics) : float
#OnPrintPage(in e : PrintPageEventArgs)
#DrawRow(in graphics : Graphics, in rect : Rectangle, in rowindex : int, in font : Font)
#DrawHeader(in graphics : Graphics, in rect : Rectangle)
#DrawFooter(in graphics : Graphics, in rect : Rectangle, in pagenumber : int)
+Print()

MyLibrary.Controls


Forms::Panel
Graph
-components : Container = null
-axiswidth : ushort = 70
-titleheight : ushort = 30
-margin : ushort = 10
#majortickwidth : ushort = 10
#minortickwidth : ushort = 5
-ticklabelwidth : ushort = 50
-xaxislabelgap : ushort = 30
-majoryinterval : short
-minoryinterval : short
+PrintDocument : PrintDocument = new PrintDocument()
-series : GraphPoint[]
-title : string
-xaxislabel : string
-yaxislabel : string
+Graph()
#Dispose(in disposing : bool)
-InitializeComponent()
#OnPaint(in pe : PaintEventArgs)
-PrintPage(in sender : object, in e : PrintPageEventArgs)
#Draw(in graphics : Graphics, in rect : Rectangle)
-DrawTitle(in graphics : Graphics, in rect : Rectangle)
#DrawXAxis(in graphics : Graphics, in rect : Rectangle)
#DrawYAxis(in graphics : Graphics, in rect : Rectangle)
#DrawPlotArea(in graphics : Graphics, in rect : Rectangle)
+Series() : GraphPoint[]
+HighValue() : int
+Title() : string
+XAxisLabel() : string
+YAxisLabel() : string
+MajorYInterval() : short
+MinorYInterval() : short
-series
1
*
«struct»GraphPoint
+Name : string
+Value : int
+GraphPoint(in name : string, in value : int)
Forms::ListView
EditListView
-components : Container = null
-editbutton : Button
-removebutton : Button
+EditListView()
#OnSelectedIndexChanged(in e : EventArgs)
+EditButton() : Button
+RemoveButton() : Button
#Dispose(in disposing : bool)
-InitializeComponent()
LineGraph
-components : Container = null
#DrawXAxis(in graphics : Graphics, in rect : Rectangle)
#DrawPlotArea(in graphics : Graphics, in rect : Rectangle)
+LineGraph()
#Dispose(in disposing : bool)
-InitializeComponent()
-DrawGrid(in graphics : Graphics, in rect : Rectangle, out xinterval : float, out yinterval : float)
Forms::ComboBox
ImageComboBox
-components : Container = null
-imagelist : ImageList
+ImageComboBox()
#Dispose(in disposing : bool)
-InitializeComponent()
+ImageList() : ImageList
#OnDrawItem(in e : DrawItemEventArgs)
1
*
ImageComboItem
-text : string
-imageindex : int = -1
-tag : object
+ImageComboItem(in text : string)
+ImageComboItem(in text : string, in imageindex : int)
+ImageComboItem(in text : string, in imageindex : int, in tag : object)
+Text() : string
+ImageIndex() : int
+Tag() : object
+ToString() : string