Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 About this document

This document has been written during a diploma thesis. It is the main part of the whole documentation and describes the core of this work. It gives a project overview and discusses all technical aspects.

1.2 Introduction

During 2003, three different versions of an order application were developed. This applications represent different approaches to deal with databases. The application itself is not very complicated. It manages persons including addresses who can make orders. The respective database and application designs are discussed later in this document.

The purpose of this work originated from the third of those applications. That version manages database related tasks through an O/R Mapper. Implementing such a mapper can be very challenging and error-prone. This is where *STORM* comes into play. The idea behind it is to have a framework which generates all mapping code automatically, based on input parameters. This document describes how this can be achieved and also explains the limitation of such an approach. As a reference implementation to prove the usability of *STORM*, a fourth order application has been created. Throughout this document, all implementation examples are taken from this reference

implementation. At chapter 7 the reference application is taken as a real world example to show how to develop an application with *STORM*.

1.3 About O/R Mapping

Today, most new applications are developed by using an object oriented programming language like C++ or C#. Nevertheless, most databases have a relational structure. Relational means that relations between objects are always be modelled by defining foreign keys in tables. This structural difference is known as "type mismatch". Because of that, a mechanism is needed that provides a way to work with databases in an easy as possible way. One possibility is to use datasets. Datasets have a similar structure to that of a relational database and are built-in types in C#. This method was implemented in the first version of the order application. Another possibility is to write code which manages all the mapping. Namely this means mapping classes to tables, variables to columns, etc. This has the advantage that a programmer can code in a pure object oriented manner an does need to pay less attention to the database. For him, database tasks are done automatically. Although most O/R mapper, including *STORM*, have more features implemented, mapping the in-memory structure to the database structure is the most important task of an O/R mapper.

1.4 How to start

At the end, an O/R mapper is nothing else than code that is responsible for certain tasks. There are several methods to create this code. A catchphrase in the context of software engineering is "generic programming". Generic programming is a technique aiming at writing programs as general as possible, without sacrificing efficiency by doing overgeneralising. Another technique is called "generative programming". Generative programming enables programs to be automatically constructed. There are more programming techniques like aspect-oriented or functional programming which are not taken into account here.

Both, generic and generative programming have their strengths and weaknesses. The third order application made use of the generic programming technique. As a counterpart, this thesis should result in the knowledge of the

usability of a generative programming technique for this specific problem.

As stated before, a generative programming technique enables programs to be automatically constructed. This sounds promising, but before one can start writing a program which constructs another program, he needs to define a starting point. Transferred to the problem of O/R mapping, this means that one needs to define where to make the mapping definitions. Three scenarios would be sensible:

- The database design is the starting point. Out of this design, all domain objects and the mapping code are generated.
- The domain model is the starting point. Out of this, the database tables and the mapping code are generated.
- Both, the domain model and the database design are taken as starting point. Additionally, the mapping needs to be defined. Out of this, the mapping code is generated.

Which approach should be chosen depends on the requirements for a given project. Each has been realised in commercial and open source projects. In this thesis, we have chosen to use the second approach where the domain model should be taken as starting point. Although with this approach the database tables could be generated automatically, it is not implemented because we use the existing database from the previous order applications. This is not a problem, because the mapping can be specified in the domain objects and this works whether the database tables are generated or already exist.

Chapter 2

Requirements Overview

2.1 *STORM* external

The main advantage of using a tool like *STORM* is the time savings in project developments and the accuracy of the generated code. This in mind, the following requirements are specified:

- Mapping between the in-memory objects and the database tables is done automatically.
- Easy and understandable configuration and usage.
- There are no other dependencies than to CodeSmith, such that *STORM* can be used independently.
- Is integrated in the make process of the Visual Studio .NET.
- Can also be used without Visual Studio .NET.
- The resulting program is efficient regarding network performance.

This is one would expect from this framework. Because it is not always easy to write templates for a general purpose, it is important to keep the target in mind. A framework like *STORM* cannot cover all requirements which can occur in any project. An exactly defined boundary is needed. This boundary defines which problems can be solved with *STORM*. It is

described in the next section and in more detail in later chapters. The above requirements are of general nature and apply to every project which uses the technique of generative programming in the scope of an O/R Mapper.

2.2 *STORM* internal

These requirements are more specific and could be called "features of *STORM*". They are not directly connected with the usage of the framework but affect the design of it. This list of features could be extended in many ways but it is a good starting point:

- No dependencies between the framework, the generated code and the code which uses them (namely a client) may exist.
- A locking mechanism is implemented.
- Insert, update and delete statements are handled by the framework and executed in the right order.
- SQL code is hidden from the user and generated automatically.
- Custom finder methods can be defined.
- Custom constructors can be defined.
- Transferred objects are stored to minimize remote calls.
- Data integrity is ensured.
- A well defined set of database mapping types are supported.

Most of these requirements are not easy to implement but even though, they are very important in terms of the usability of the framework. These requirements are a starting point for the whole design. Every one will be discussed in detail in this document.

Because most of the above requirements can be solved by using and applying appropriate patterns, the next section is devoted to these patterns.

Chapter 3

Patterns Discussion

3.1 Why Patterns

A pattern describes both a problem which occurs over and over again and the core of the solution to that problem. This solution can be used whenever this or a similar problem exists. Patterns can be combined and are therefore a valuable design strategy for large applications. There are many advantages of using patterns over a classical design. The scope of this thesis is not to list all of the advantages but to talk about specific patterns which are worth studying for this purpose.

Not all of the following patterns describe a technically complex problem but they give a vocabulary to talk about problems in a manner which everybody understands easily who knows this vocabulary.

3.2 Which Patterns

There exists thousands of patterns out there. Not all patterns are relevant for a given application, in fact, a minority is. This raises the problem of choosing the patterns, which should be done carefully because it directly affects the application's design.

The following list is a summary of all patterns relevant for this thesis. They are mainly taken from M. Fowler [1]. The list is in alphabetical order.

Table 3.1: List of relevant patterns

Data Mapper (p. 8)	Metadata Mapping (p. 13)
Domain Model (p. 7)	Optimistic Offline Lock (p.16)
Factory Method (p. 18)	Query Object (p. 14)
Foreign Key Mapping (p. 11)	Registry (p. 17)
Identity Field (p. 12)	Remote Facade (p. 15)
Identity Map (p. 9)	Repository (p. 15)
Layer Supertype (p. 16)	Separated Interface (p. 18)
Lazy Load (p. 9)	Service Layer (p. 7)
Mapper (p. 17)	Unit of Work (p. 11)

3.3 Describe the Patterns

Of course, one needs to know what a pattern at the end does and which benefits it provides. First, each pattern mentioned in the last section will be described separately before they are put in a big context. The following description is not meant as a complete reference but as a short description of each relevant pattern. Especially implementation details are left aside. They will be addressed in further chapters. Event though, this description should give enough information to understand what each pattern is for. For a more detailed discussion of these patterns, please read M. Fowler's Patterns of Enterprise Application Architecture [1].

3.3.1 Domain Model

The domain model pattern essentially describes that the domain layer is organised in an object-oriented (OO) manner. This technique is commonly used and understood and therefore not described in further detail.

3.3.2 Service Layer

Enterprise applications often have different interfaces which provide access to business logic and data they store. Common interfaces are those for web-services, user interfaces, data loader, etc. A *Service Layer* defines a boundary to the application and available operations (see Figure 3.1). Therefore, it encapsulates the application's business logic. To separate business logic from

the client interfaces has several advantages. First, the interface and therefore all possible operations are well defined and relatively easy to use for clients. Second, it reduces the burden of multiple implementations of the same functionality and improves code maintainability. A *Remote Facade* (3.3.12) is an example of a *Service Layer*.

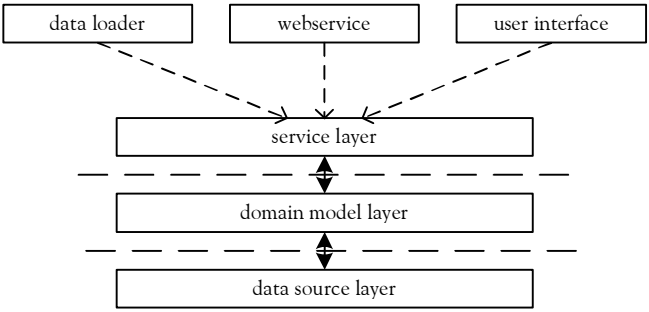


Figure 3.1: Service Layer defines application's boundary

3.3.3 Data Mapper

A *Data Mapper* is a layer between Domain Objects and a Database. A *Data Mapper* is used because Objects and a relational database have different mechanisms for structuring data. The *Data Mapper* separates in-memory objects from the database and transfers data between the two. With this pattern, the in-memory objects do not need to know about the database at all. Figure 3.2 shows the general idea of a *Data Mapper*.

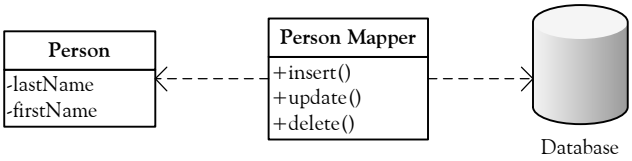


Figure 3.2: Data Mapper Overview

The whole layer of *Data Mapper* can be substituted, either for testing purposes or to allow a single domain layer to work with different kind of databases. This becomes very powerful if a *Data Mapper* is combined with other patterns. This topic will be addressed in later chapters.

3.3.4 Identity Map

The basic idea behind *Identity Map* is to have a series of maps containing objects that have been pulled from the database. If you load an object from the database, you first check the maps. If there already is an object which corresponds to the one you are loading, you do not need to load it. This is illustrated in Figure 3.3

How many maps are needed depends on the application. You can either choose one map per class or one map for the whole session.

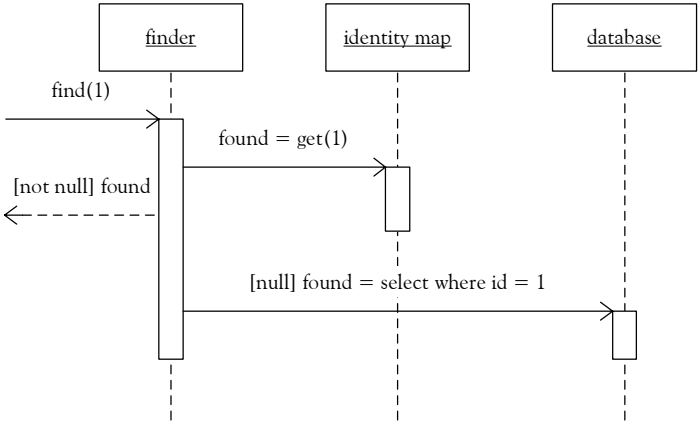


Figure 3.3: Mapping a collection to a foreign key

3.3.5 Lazy Load

If you load an object from the database into memory it is often useful to load objects that are related to it. This makes working with objects easier, because you don't have to load every single object yourself. The drawback

of this mechanism is that it can have the effect of loading a huge number of related objects. This in turn can hurt the performance of your application drastically, especially when only a few objects are actually needed. *Lazy Load* interrupts this loading process and leaves a marker in the object structure so that it can be loaded only when it is used. This is shown in Figure 3.4.

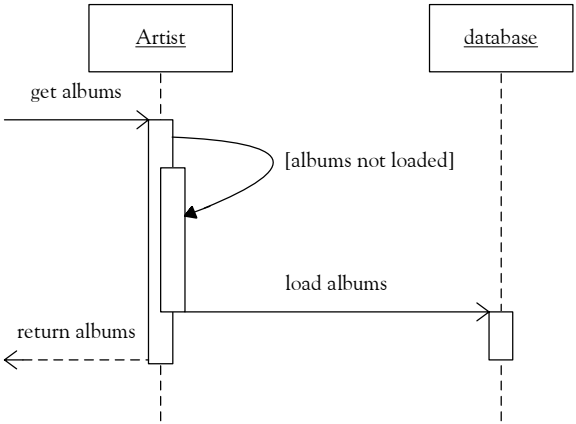


Figure 3.4: Sample Lazy Load Sequence

Using *Lazy Load* does not always improve performance. For instance, if you need to load all related objects, Lazy Load can even make performance worse that it was without it.

There are four ways for implementing *Lazy Load*

- Lazy Initialisation
- Virtual Proxy
- Value Holder
- Ghost

All of this methods have their own benefits and drawbacks but for the purpose of this thesis, a *Virtual Proxy* pattern seems the most powerful and appropriate. A *Virtual Proxy* is an object that looks like a real object but does not contain any data. Only when one of its methods is called does it

load the correct object from the database. This pattern is described in E. Gamma's Design Patterns [2].

3.3.6 Unit of Work

It is important to know what you change in your domain model in order to apply those changes to the database. A *Unit of Work* keeps track of what you have done during a business transaction and figures out what needs to be done to get the database in a consistent state. Therefore, a *Unit of Word* needs to be informed every time an object is change, created or deleted. There are different ways to implement this. Either the user of an object register the object with the *Unit of Work* and mark objects as dirty after making any changes or the *Unit of Work* handles this by its own. The latter case is preferable because it is less fault-prone. Figure 3.5 shows the working mechanism of this pattern.

3.3.7 Foreign Key Mapping

In an object-oriented system, objects are connected to each other directly or by object references. Relational databases use foreign key references for this. To save an object to a database, it is important to save these object references and map them to foreign key references on the database. This is, what a *Foreign Key Mapping* pattern does, it maps an object reference to a foreign key.

These mappings can be of different kind and complexity. The simplest case is, if two objects are linked together. This association can be replaced by a foreign key in the database. This is shown in Figure 3.6.

Things get more complicated when one has to keep track of a collection of objects, because a collection cannot be saved to a database. The direction of the reference needs to be reversed. In our example, which is shown in Figure 3.7, this means to store a foreign key of the album in the track record.

This pattern needs to be discussed in more detail, especially cases where one needs to update a collection objects. This will be explained in the appropriate section, where implementation details are subjected.

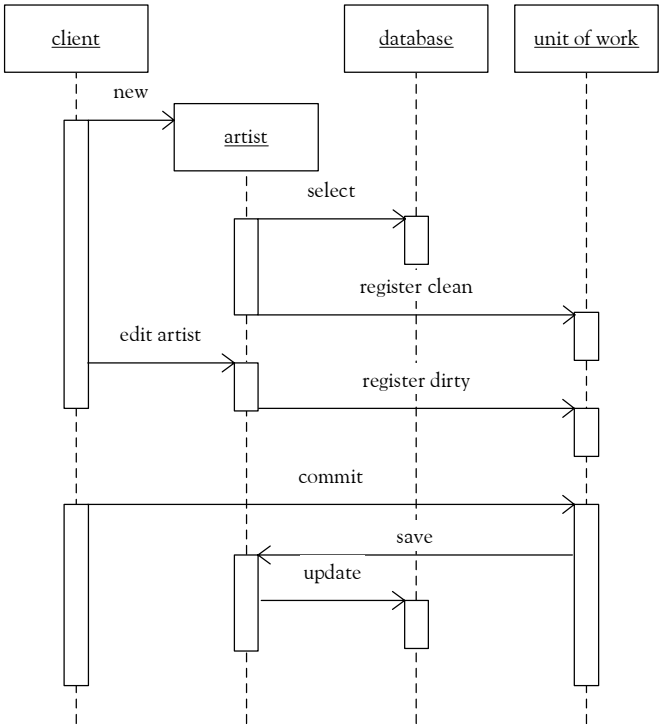


Figure 3.5: Unit of work mechanism

3.3.8 Identity Field

A relational database distinguishes one row from another by using keys. A key usually is nothing else than a `long` data-type. However, in-memory objects do not have such key (it is the systems responsibility to keep track of object identities). Reading from a database is never a problem where database keys are concerned, writing is a problem though. One needs to save a copy of the database key in the in-memory object to tie this object to the corresponding database row. The *Identity Field* pattern should be used whenever a mapping between in-memory objects and rows in a database is needed (which is almost always).

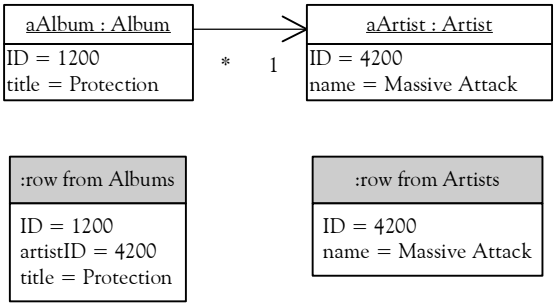


Figure 3.6: Simple Foreign Key Mapper

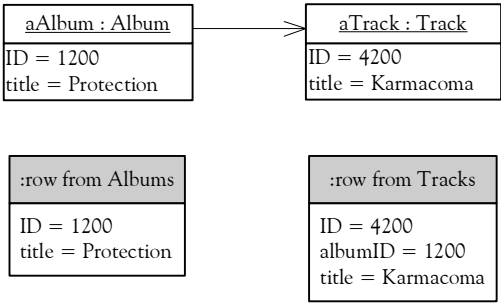


Figure 3.7: Mapping a collection to a foreign key

3.3.9 Metadata Mapping

As already mentioned, in-memory objects and database rows have different mechanisms to organise and structure data. Mapping this structures is what object-relational Mapping is all about. To achieve this, many lines of code are needed to describe how fields in the database correspond to fields in in-memory objects. The resulting code is repetitive to write. With a code generation tool (in this case OOXGen - which is a result of this thesis) a *Metadata Mapping* pattern can be used to generate those repetitive lines of code. The input of the OOXGen is the metadata and the output is the source code of classes that do the mapping. On most occasions the metadata

is kept in a separate file, such as an XML file or even in the database itself. Figure 3.8 shows the principle of the *Metadata Mapping* pattern.

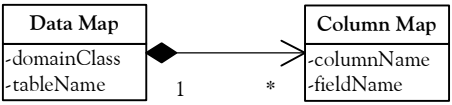


Figure 3.8: Holds details of object-relational mapping in metadata

3.3.10 Query Object

A *Query Object* is a structure of objects that can form itself into a SQL query. A query can be formed by referencing classes and fields rather than tables and columns. This allows a client to form a query of various kinds and turn those object structures into the appropriate SQL statement. With *Query Object* you can separate SQL statements from the rest of the code which makes it possible to use different objects for different database schemata. This can be very powerful and makes code more readable. Furthermore a developer needs to know neither the details of the database nor he needs to be familiar with the SQL language. Figure 3.9 shows an example of a possible *Query Object*.

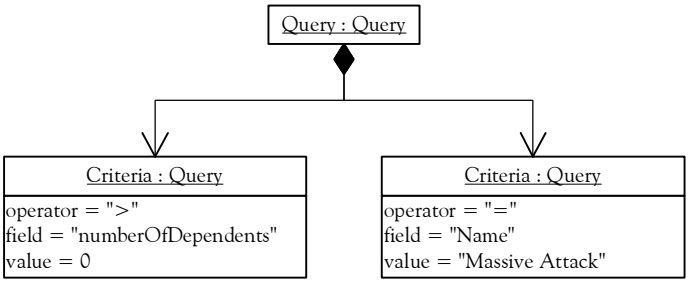


Figure 3.9: An Object Query representing a database query

3.3.11 Repository

A *Repository* is a separation Layer between domain objects and a *Data Mapper*. It isolates domain objects from details of the database access code. Conceptually, a *Repository* encapsulates the set of objects persisted in a data store and the operations performed over them. It presents a simple interface for clients which can create a Criteria object specifying the characteristics of the objects they want returned from a query. These Criteria Objects are submitted to the *Repository* for satisfactory. In conjunction with *Query Object*, this pattern adds a large measure of usability to the object-relational mapping layer. Client code do not need to use a *Query Object* directly, instead it creates criteria objects and passes them to the *Repository*, asking it to select those objects that match.

3.3.12 Remote Facade

A remote call is always very expensive in terms of performance. It is better to do things in a single call rather than multiple calls. This is opposed to local calls. A *Remote Facade* provides a coarse-grained interface for remote calls. This principle is illustrated in Figure 3.10. Rather than making a remote call for each element in an address, one call for all elements is performed. Therefore, this pattern describes the principle of designing coarse-grained interfaces which remote clients, such as web-services, can use. This pattern is a special *Facade* which is described in E. Gamma's Design Patterns [2]. A *Facade* generally provides a simplified, standardised interface to a large set of objects.

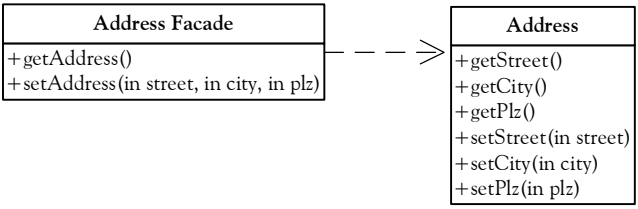


Figure 3.10: A coarse-grained remote call

3.3.13 Optimistic Offline Lock

A business transaction often includes a series of transaction. A single transaction is locked by the database manager. That is, no other transaction can access the same data at the same time. When you have more than a single transaction, you need a mechanism to ensure that data at the end of a business transaction is in consistent state. This can be done with locks. There are two different approaches how a transaction can obtain a lock. This is either pessimistic or optimistic offline lock. The main difference is that *Pessimistic Offline Lock* assumes that the chance of session conflict is high whereas *Optimistic Offline Lock* assumes that the chance of one transaction conflicts with another is low. *Pessimistic Offline Lock* is easier to implement but it limits system concurrency. This is because at a given time only one user can work with the locked data. *Optimistic Offline Lock* allows multiple users to work with the same data at the same time. This is often very desirable but means to solve some problems which can occur when multiple users change data at the same time. Namely this problems are with `UPDATE` and `DELETE` SQL statements. Additionally, inconsistent reads must be detected.

An *Optimistic Offline Lock* is obtained by validating that, in the time since a transaction session loaded a record, another session has not altered it. This is usually done by comparing a version number stored in a session with the version in the record data. Figure 3.11 shows an `UPDATE` with *Optimistic Offline Lock*. The version number is checked within the `UPDATE` statement and a row count of 1 is returned in case of success otherwise a row count of 0 is returned. An example of *Optimistic Offline Lock* is the concurrent versions system (CVS).

Checks for inconsistent reads can become very tricky even with a few transactions. A solution can be to group together transactions which depend on each other. This group can be treated as a single lockable item.

3.3.14 Layer Supertype

The *Layer Supertype* pattern describes the idea of an interface. An interface acts as a superclass which defines all common methods. Each class which references this interface independently implements these methods.

This way of implementing common features is very widespread in object-oriented programming and is therefore not explained in more detail.

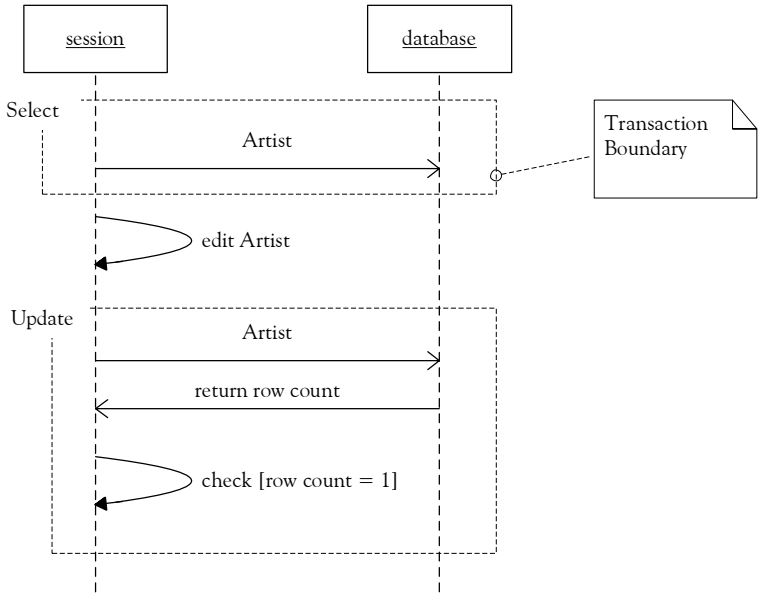


Figure 3.11: UPDATE with Optimistic Offline Lock

3.3.15 Mapper

A *Mapper* is a layer between subsystems. It decouples these subsystems from each other. When using a *Mapper*, either subsystem is not aware of the other but they still can communicate through the *Mapper*. The most common case of a mapping layer is a *Data Mapper* (3.3.3).

3.3.16 Registry

A *Registry* is essentially a global object. If you have methods for which you do not have a reference to start with, you can register this methods in a *Registry*. A *Registry* is usually implemented as a singleton class. This makes it possible to get access to those methods from within the whole program. There are always workarounds in case you do not like to use a *Registry*.

3.3.17 FactoryMethod

A *Factory Method* defines an interface for creating an object. Which object is to be created is decided by subclasses. Usually a *Factory Method* is needed when a class cannot anticipate the class of objects it must create. This is often the case with frameworks which must provide a mechanism to create concrete implementations of abstract classes. A framework does not know at design time which classes this will be. The idea is to pass all needed information to the *Factory Method* which in turn will create the object. The advantage is to eliminate the need to bind application-specific classes into the code. The design of a *Factory Method* is shown in Figure 3.12.

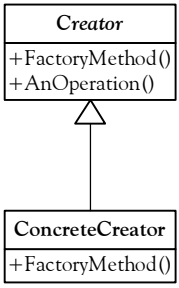


Figure 3.12: Factory Method Design

3.3.18 Separated Interface

A design paradigm is to decoupling individual parts of the system. A way to achieve this is to group classes in packages and control the dependencies between them. *Separated Interface* is a way to manage these dependencies. You can define an interface in one package and put the implementation of this interface in a different package. This way, you do not have to reference classes in a different package, you only need a reference to the interface. A sample of this pattern is shown in Figure 3.13

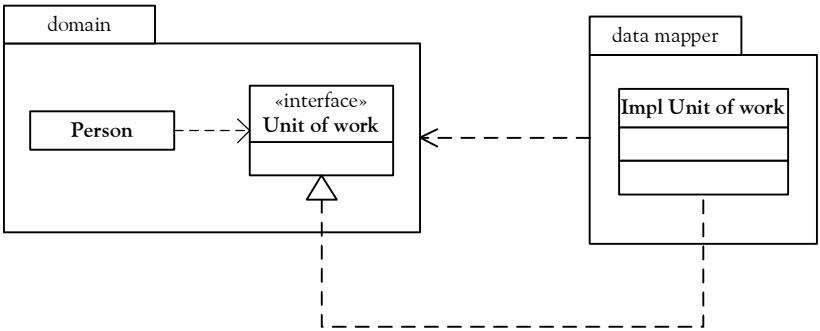


Figure 3.13: Sample of Separated Interface

Chapter 4

Design Considerations

4.1 Fundamental Approach

STORM is a framework to generate code for an object-relational mapper on basis of templates. There are a few question which need to be answered before starting a design for the framework. These questions are:

1. What needs to be parameterised?
2. How can parameters be defined?
3. How can parameters be passed to templates?

4.1.1 What needs to be parameterised?

Obviously, for an object-relational mapper the most important thing to parameterise is the mapping. This means, the database structure, the structure of the in-memory objects and the mapping between them must be defined. For example, one needs to be able to define that an in-memory object of a class `Person` maps to a database table called `Persons`. Not only the structure must be configurable but also primary keys, foreign keys and relations between tables.

4.1.2 How can parameters be defined?

After we know *what* needs to be parameterised, the question is *how*. At the beginning of this thesis, three different fundamental approaches were analysed and tested.

The first approach is XML Schema: There are two XML Schema files and one plain XML file. One XML Schema file is to define the in-memory object structure, the other one is to define the database structure and the XML file is used to describe the mapping between the two. The problem of this approach is its complexity and unmanageability.

Another approach was an extended C# compiler. This compiler would work with custom defined keywords. This keywords could be used to define the structures and mapping.

The third approach to describe the mapping is to write an abstract class with custom attributes. A code snippet of such a class is shown in Listing 4.1. After compiling this has been compiled, the custom attributes can be read using reflection from the resulting dll. With these attributes, all relevant information can be retrieved.

The main advantage of this is that a programmer can define everything in a single file (single-source approach) rather than multiple files. This means, not only the mapping code could be generated automatically, but also the corresponding database tables. Furthermore it is a natural way for programmers because they are used to write classes.

One could say that a drawback of attributes is that they cannot be changed at runtime. This is true but also, it is not required. Every information given in this abstract class is static within the scope of an application.

From our point of view, this approach is the most feasible.

4.1.3 How can parameters be passed to templates?

Once we know how to parameterise everything we need a way to let the template know all these parameters. The first thing which is given is that custom attributes in C# are bound to the class in in which they are defined. Therefore, attributes must be read by using reflection. First, we could generate code which makes use of reflection or second, we use reflection in the template itself an generate code containing no reflection but the concrete code resulting from the custom attributes. The latter case is more according to the idea of generative programming. If we used reflection in the generated

Listing 4.1: Abstract class with custom attributes

```
[Table("Addresses", true),
VersionField("chTimeStamp"),
GenerateCode]
public abstract class Address : DomainObject
{
  [Factory]
  public abstract class AddressFactory
  {
    public abstract Address createAddress(
    [ParameterDef("City")] string city,
    [ParameterDef("Street")] string street,
  }

  [Column("AddressID"),
  PrimaryKey]
  public abstract int AddressId {get;}

  [ToMany(typeof(OrderDetail), "OrderId")]
  public abstract IList OrderDetails {get;}

  [Column("PersonID"),
  ToOne(typeof(Person), "PersonId")]
  public abstract Person PersonRel {get; set;}

  [Column("City")]
  public abstract string City {get; set;}

  [Column("Street")]
  public abstract string Street {get; set;}
}
}
```

code the application's performance would be poorer. What is more, it would not make much sense because we would not need to write any templates at all.

4.2 Object Dependencies

Whenever code is generated, it is important to design the application carefully. An important consideration must be object dependencies. This is, an abstract class or any class written in the client code may never depend on a generated class nor may a generated class depend on another generated class. If such a dependency existed, it would result in a dependency cycle which cannot be resolved. Consider an abstract class `Person` which results

in the concrete implementation class `PersonImpl`. Further, consider a client which wants to set the property *Name* for a `Person` and uses the concrete implementation `PersonImpl` for this (see Figure 4.1). This would have the effect that the client class could not be compiled before the concrete implementation is generated. But to generate the concrete implementation, the client code needs to be compiled (because the resulting dll is needed). This could be avoided by putting the client code in a separate project. But at the latest when you have dependencies between abstract classes and concrete implementation classes this cannot be avoided.

Therefore a mechanism is needed that the client code just need to reference the abstract class but still can set the property in the concrete implementation object. For this we use a *Factory Method* which provides us with a concrete implementation without the need of referencing it directly. How exactly the *Factory Method* was used is explained in 6.4.

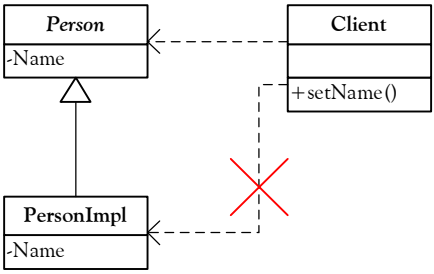


Figure 4.1: Dependencies between classes

4.3 Object Lifetime

An object which has been transferred from the database can be kept for further use in an *Identity Map* (see 3.3.4). The question is, how long this object resides in this map. An *Identity Map* is stored in a *Unit of Work* (see 3.3.6) and therefore, if the *Unit of Work* object is destroyed, the *Identity Map* and all objects stored in it will be destroyed too. This is indicated in Figure 4.2. This brings up the question how long a *Unit of Work* object should exists. There are two answers for that:

- The objects lifetime is as long as a user session exists. This is, a *Unit of Work* object is created on a per thread basis. A new thread is started on each new session.

- The object is on a per transaction basis. This is, a new *Unit of Work* object is created on every commit.

These two methods represent completely different concepts. The first implies that it is possible for an object to reside in an *Identity Map* for a long time. On the one hand, this increases the risk of consistency errors but on the other hand, it follows more the idea of an *Optimistic Offline Lock* (see 3.3.13) architecture. This means that consistency errors will rarely occur. Additionally, we avoid consistency errors by using a lazy load system which checks if the object in question is up to date (see 6.7.2).

The second method is more adequate for a system with many users who are likely to edit the same object simultaneously. In such systems, consistency errors are likely to occur. Whenever this is the case, it makes not much sense to store objects in an *Identity Map* for a long time to just remove them upon the next update.

We assume that the first case is more appropriate in most cases and use therefore a per session based *Unit of Work*.

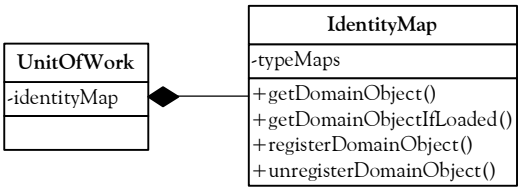


Figure 4.2: Relation between *Unit of Work* and an *Identity Map*

4.4 Locking Mechanisms

Depending on the application, different database locking mechanisms are appropriate. There are two main ways how database locking can be implemented:

- The whole database is locked during a transaction.
- Only certain records of the database are locked during a transaction.

The first method its easier to implement and adequate for many applications. It locks the whole database within a business transaction. The disadvantage of this is that whenever several people access the same data within a business transaction, only one of them can commit without a conflict. For all others, the transaction will fail.

The second method solves this problem by using more sophisticated locks. Several strategies can be used:

exclusive write lock A business transaction acquires a lock in order to edit session data. Reading these data is still possible.

exclusive read lock A business transaction is required to acquire a lock simply to load the record.

read/write lock A record cannot be write-locked if any other business transaction owns a read lock on it and vice versa. Concurrent read-locks are acceptable.

For this thesis, we use the first method which locks the whole database. The reason is that we do not have a concrete application for which a more complicated locking mechanism would be needed. But Generally, it is important to choose the right locking strategy.

4.5 Mapping Types

Not all possible mapping types are supported by *STORM*. This section describes the supported types. That is, which elements of the object oriented code can be mapped to which database elements. The custom attributes which define the mapping is named for each type. A detailed description of the attributes can be found in section 6.8.

Class A class is always mapped to one table. Even though, it would be possible to map more than one class to the same table.

The Table attribute is mandatory for each class.

Member variable A member variable of a class is mapped to a column within the corresponding table.

The Column attribute sets the name of the column.

One to one relation A relation from one table to exactly one entry of another table. Table A contains the primary key or a unique key of an entry in table B. In the source code, this mapping is done by referencing object A to object B.

ToOne is the attribute defining this relation.

One to many relation A relation from one table to several entries of another table. A database handles this relation by placing a foreign key in each entry in table B. With this foreign key, entries in table B know that they belong to table A.

In the source code, object A has an array list which contains all related objects B. *STORM* needs a back reference (to-one relation) in class B for each to-many relation in class A.

ToMany is the attribute defining this relation.

Many to many relation To realise such a relation on the database, an linking table is needed to store the relations between entries.

In the code it is possible to manage many-to-many relations with array lists without the need of an extra object between them.

We do not support this special case. To make a many-to-many relation it would be necessary to make an extra object which maps to the linking table.

Enumeration/Lookup tables Enumerations are not needed and often not supported by databases. If you are thinking about using an enum you should consider to save the values in a lookup table. Enumerations are not supported by *STORM*.

Chapter 5

Code Generation

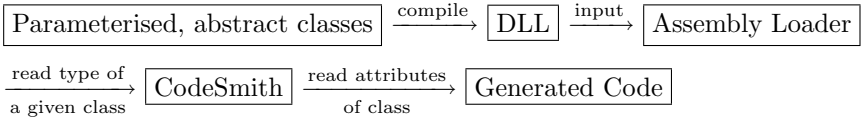
5.1 Finding a Tool

Mainly two projects exist which are able to generate code on basis of templates. These are Workstate's Codify[1] and CodeSmith[2]. Both projects were evaluated at the beginning of this thesis. CodeSmith has several advantages over Codify. First, it is open source and it seems that it has a bigger community. Moreover, it cannot only be integrated in Visual Studio .NET, it has a command line tool and it can be run from the CodeSmith standalone program. Particularly the command line tool is useful for creating makefiles. Furthermore, CodeSmith has a much more powerful way to write custom templates. Therefore, the decision clearly felt on CodeSmith.

5.2 Procedure of Code Generation

To be able to customise CodeSmith templates, one needs parameters as input for them. Input parameters are in the case of *STORM* the custom attributes in the abstract classes. The procedure to actually generate the code is as follows:

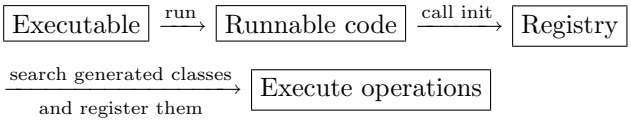
[1] http://www.workstate.com/codify
[2] http://www.ericjsmith.net/codesmith/



First, all abstract classes must be compiled. The resulting DLL is used as input for the `AssemblyLoader` which is part of *STORM*. The `AssemblyLoader` loads the assembly and searches for a specified class. If the `AssemblyLoader` finds the specified class in the assembly, it returns the type definition for it to CodeSmith. CodeSmith in turn takes this type definition and reads its custom attributes. On the basis of these attributes, the template code is executed and the resulting code is generated.

5.3 Running the code

When all concrete classes have been generated, the code can be executed. To execute the code, the first step is to initialize the `Registry`. This registry takes the runnable code as argument and searches for generated classes in it. Every class which is found is registered in a hashtable. Furthermore, the `Registry` operates as a *Factory Method*. The procedure looks as follows:



Chapter 6

Concepts

6.1 Packages

The software is split into two parent packages which themselves are split into sub packages. Figure 6.1 shows those packages. One of the main package is `HsrOrderApp` comprising of everything from the sample order application. This is specific code, including generated classes. The second main package is `STORM` and comprises of code which is not specific to the sample implementation. This makes it possible to distribute the `STORM` code independently.

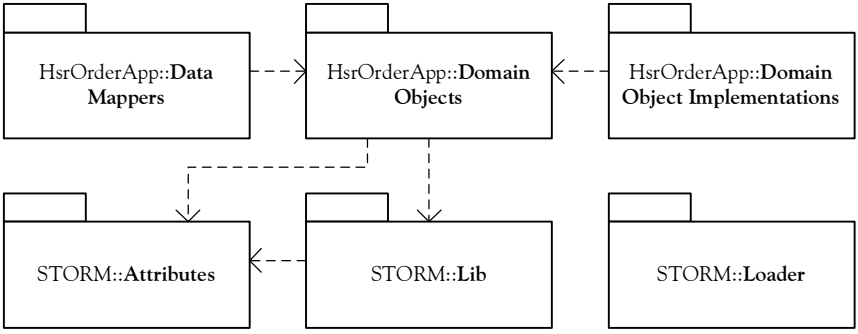


Figure 6.1: Software Packages

The `HsrOrderApp::Domain Objects` package contains all abstract classes. These classes are needed to generate the mapping code. This is the starting point to write an application. The other two packages in `HsrOrderApp::` contain only generated code. This code is split into implementations of the domain objects and the mapping code for each domain object.

The more important packages are `STORM::`. The `::Attribute` package contains all custom attributes. This includes attributes which are used internally only by *STORM* and attributes which can be used to attribute abstract classes. All these attributes are listed in section 6.8. The package `::Lib` contains all generic classes which *STORM* provides. At last, the `::Loader` package contains the `AssemblyLoader` and other classes related to CodeSmith.

The `STORM::Lib` package is the core of the *STORM* framework. Figure 6.2 shows a conceptional model of this package. Some classes are generic and only used by the classes in the domain object. In the following sections, the different concepts are described.

6.2 Registry

As soon as an application starts, it must call the `Registry`'s `init()` function. This function searches through all assemblies of the currently executed program and stores the type information of all domain object and mapper implementations in hashtables. These hashtables are used later to retrieve the appropriate implementation or mapper for a given type. Because we cannot know the type of implementations and mappers at design time, only interfaces are returned. This interfaces can be casted by the caller. The usage of the `Registry` is described in the following sections. Figure 6.3 shows the registry class and the interfaces which are used in this context.

6.3 Unit of Work

A key concept in *STORM* is the use of a *Unit of Work* pattern. This pattern has already been described in section 3.3.6. A UnitOfWork manages database connections and is responsible to execute database operations in the right order.

Every time a new object is created, that object registers itself in the

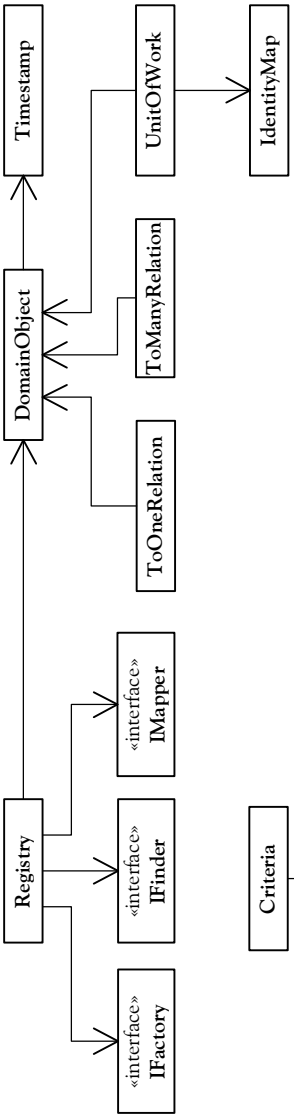


Figure 6.2: Conceptional Model of the STORM::Lib Package

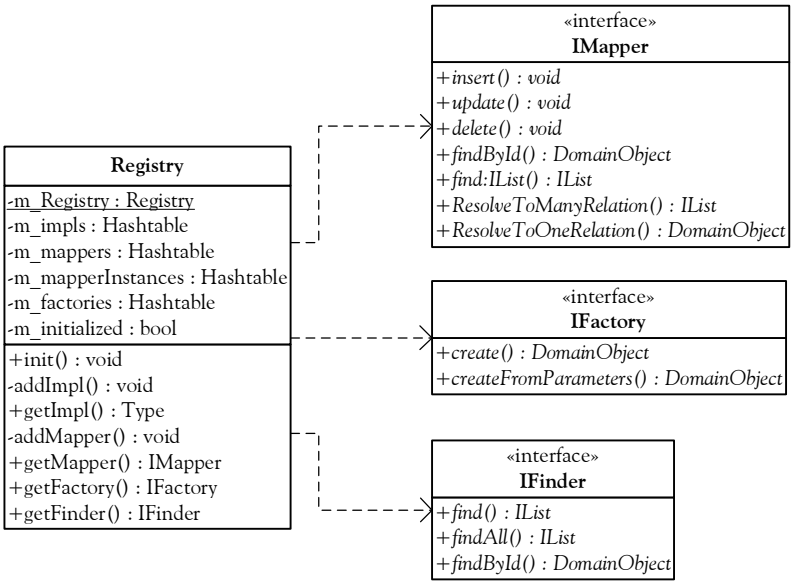


Figure 6.3: Registry and depending Interfaces

UnitOfWork as new. Similarly, if an object has been changed or deleted, that object registers itself as either dirty or removed. The UnitOfWork stores all this and executes the appropriate operations in the right order upon a commit request. If an object is transferred from the database, this object is registered as clean.

Connections are handled through a `Transaction` object. Every time the UnitOfWork receives a commit request, it starts a new transaction. After all commands are executed, the transaction is closed. This is related to the locking mechanisms described in section 4.4. Also the lifetime of a UnitOfWork has already been described in section 4.3.

6.4 Factory

As mentioned in chapter 4, it is important to avoid dependencies between generated and non-generated code. The solution is to use the *Factory Method* pattern. In the case of *STORM*, this means that whenever a new object is created, it must be handled through the `Registry`. This is somewhat more complicated but the most proper solution for it. For example, if a client wants to create a new `Person`, it needs a constructor for it. If this constructor was declared in the abstract class, it would be of no use because abstract classes cannot be instantiated. And because of the dependency problem, a constructor in the domain object implementation cannot be called directly. But again, in order to create a new `Person`, we need to call a constructor.

The solution is to declare an internal class in the abstract domain object class and to parameterise it with the Factory attribute. This, for a `Person`, would look like Listing 6.1.

Listing 6.1: Defining a Factory Class

```
public abstract class Person : DomainObject
{
  [Factory]
  public abstract class PersonFactory
  {
    public abstract Person createPerson(
      [ParameterDef("PersonName")] string name,
      [ParameterDef("Password")] string password);
  }
}
```

Out of this declaration, a `FactoryImpl` and a static Property are generated in the domain object implementation. When now a `Person` object should be created, the client can call the `Registry` for an appropriate FactoryImpl object. On this FactoryImpl object, the declared constructor can be called. Such a call would look like:

```
Person newPerson =
  ((Person.PersonFactory)Registry.Instance.getFactory(typeof(Person)))
  .createPerson("Jack", "password");
```

With this mechanism, there is no dependency between object which should not be. The drawback is that the usage is somewhat more complicated.

Even if the Factory declaration in the abstract class is omitted, a FactoryImpl and a static Property will be generated in the domain object imple-

mentation. This is because the `IFactory` interface declares two methods to create an object. These methods can be called without declaring a custom Factory class in the abstract class. Furthermore, they are needed internally by *STORM* to create objects which has been loaded from the database but do not exist as in-memory objects.

6.5 Finder Methods

6.5.1 Self Defined

Another problem occurs with finder methods. Depending on the object in question, the finder methods will look differently. For example, a client wants to be able to call a method which looks like:

```
public IList findByNameAndPassword(string name, string password){...}
```

This would return a list containing `person` objects which match the given name and password. The question is where to declare this methods. The solution is similar to the factory described in the last section. Finder methods can be declared in the abstract class as methods of a class attributed with the Finder attribute. An example of such a declaration is given in Listing 6.2.

Listing 6.2: Defining a Finder Class

```
public abstract class Person : DomainObject
{
  [Finder]
  public abstract class PersonFinder
  {
    public abstract IList findByName([ParameterDef("Name")] string name);
    public abstract IList findByNameAndPassword(
      [ParameterDef("Name")] string name,
      [ParameterDef("Password")] string password);
  }
}
```

Now, if a client wants to call a finder method, it asks the registry for an appropriate finder implementation. On the returned object, the method can be called. Such a call would look like:

```
ArrayList foundPersons = new ArrayList();
foundPersons = ((Person.PersonFinder)Registry.Instance.getFinder(typeof(Person)))
               .findByName("Jack");
```

Because a finder method must be independent from a specific object (we do not want to create an object just to be able to call a find method), the implementation differs from the factory implementation. To be more precise, the code for a finder method must be in a appropriate mapper implementation, e.g. in the `PersonMapper`. As we can see later, this is only true for generic finder methods. The reason is the following problem: A client cannot call a method which is defined in the abstract class and implemented in the mapper implementation. That is because a mapper implementation is not inherited from the abstract class (in contrast to the domain object implementations). Therefore, an implementation of the finder methods must persist in the domain object implementation such that a client is able to call it.

The next consideration is how to call the generic finder method in the mapper implementation from within the domain object implementation. The solution is to use a *Query Object* pattern. With a query object, every custom defined finder method can be converted to an appropriate query object which then can be passed to a generic finder method in the mapper implementation. Generic finder methods are described in 6.5.2.

Figure 6.4 is a sequence diagram showing this finding scenario.

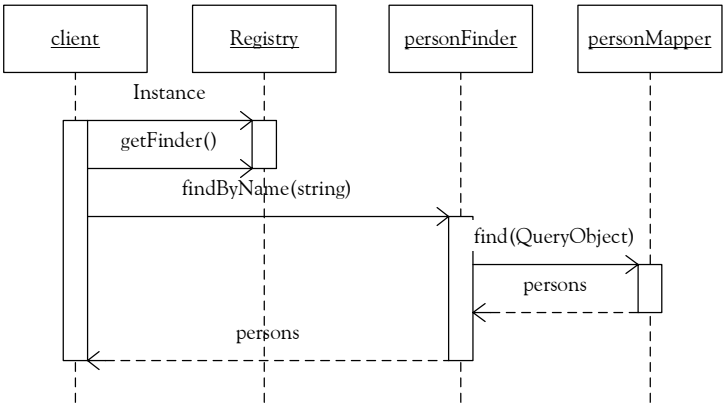


Figure 6.4: Finding objects with custom finder methods

6.5.2 Generic

Because some finder methods are known and can apply to every type of object, not all finder methods must be self defined. Those methods which are generic have been made accessible through an interface (`IFinder`). The implementation of these methods is in the mapper implementation class. The following methods have been declared in the `IFinder` interface:

```
IList find(QueryObject qo);
IList findAll();
DomainObject findById(Key id);
```

The first method takes a `queryObject` as parameter. This object holds any number of `criteria` objects. If multiple `criteria` objects are added to a `queryObject`, they are linked together with "AND". To stay with our person example, a call to find every person whose name is 'Jack' would look like:

```
QueryObject qo = new QueryObject();
qo.addCriteria(new Criteria(Criteria.Operator.Equal, "Name", "Jack"));

ArrayList foundPersons = new ArrayList();
foundPersons = Registry.Instance.getFinder(typeof(Person)).find(qo);
```

These makes it possible to form self defined find constructs through a *Query Object*. The result is the same as to define a method in the abstract class but sometimes it is more flexible to use. Figure 6.5 shows the `QueryObject` class and its relation to `Criteria`.

The other two methods are self explaining and can be used accordingly.

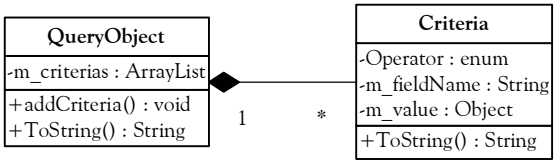


Figure 6.5: Relation between QueryObject and Criteria

6.6 Adder Methods

Adder methods are those which can be used to add related objects to the current object. For example, a `person` can have an adder method to add `address` objects which are related to the person (e.g. her/his addresses). This is different from creating and finding objects because an adder method must be called on the object itself, e.g. on a `person` object. This means, an object must be existing. Therefore, the only sensible place to have adder methods implemented is the domain object implementation.

Adder methods are used in conjunction with ToMany relations. A code snippet from the abstract class `Person` is given in Listing 6.3. The example defines a ToMany relation and a belonging adder method.

Listing 6.3: Defining an Adder Method

```
[ToMany(typeof(Order), "Person")]
public abstract IList Orders {get;}

[Adder("Orders", "Person")]
public abstract void addOrder(Order o);
```

The code for all adder methods is generated into the domain object implementation. For example, the resulting code for the adder method shown in 6.3 looks like (exception handling is omitted):

Listing 6.4: Defining an Adder Method

```
public override void addOrder(Order o)
{
  if(o != null)
  {
    o.Person = this;
    m_Orders.Add(o);
  }
}
```

m_Orders is an object of type ToManyRelation which holds all related objects. Adder methods could be generated automatically, but again, if they were, it would not possible to call those methods from a client. o.Person is the back reference from an order to a person.

6.7 Data Flow

This section is about the work flow of *STORM*. It covers and explains the most important flow scenarios. These are creating new objects, deleting objects and lazy load.

6.7.1 Create New Object

A new object is represented in an application with a new instance of a class. This instance is registered within the *Unit of Work* as new. Whenever the *Unit of Work* receives a commit, this new class needs to be written to the database. For this, a new transaction is created after the commit is received. A transaction represents a database connection for a certain, limited time. In this case, a transaction is created at the beginning of the commit method and is closed at its end. For the time of this transaction, the database is exclusively locked. To actually write the new created object to the database, the specific mapper's `insert()` method is called. This method writes the object to the database. The whole process is illustrated in Figure 6.6.

6.7.2 Lazy Load

Our object/relational mapper not only handles the mapping between classes and tables etc., it provides a mechanism to enhance performance. Vital to every application which transfers object over a network is to limit the number of remote calls. *Lazy Load* provides a mechanism to handle this need. When talking about *Lazy Load*, a few decisions need to be made before implementing a concrete solution. The most important concerns object lifetime. There are 3 main ways which are as follows:

1. Whenever an object needs to be loaded, it will be fully loaded from the database, regardless of the fact if it had been already loaded before.

2. Before loading an object from the database, a class (see Identity Map 3.3.4) which holds already transfered objects in a map is called to check if the object had been transferred before. If so, the desired object is loaded directly from this map instead of transferring it again. This sounds like a good alternative to the first mechanism but has the disadvantage of possible consistency errors. This leads to conflict resolution strategies which can be quite challenging to solve.

3. The third possibility is close to the second one except that an object's version field is compared with the version field on the database to check if the object is still up to date. If those versions match, the object from the map is loaded, otherwise the object will be reloaded from the database.

The first strategy is the most simple to implement. It does not need an *Identity Map* nor *Lazy Load*. Event though, it is not the most performant solution because objects are transferred more often. The second strategy would be, depending on the application though, more performant but much more complicated to implement. Additionally, it is not a good solution for code generation because conflict resolution strategies are very specific and can therefore not be generated automatically. For these reasons, we use the third strategy which should result in a good performance and a secure way to update objects. The process is illustrated in Figure 6.7. It shows a find operation where all addresses for a person are searched.

6.7.3 Delete Object

Another scenario originates when an object (e.g. a Person) is deleted. It is not possible to just delete the person on the database because a person can have relations like addresses which has to be deleted first. In our example, this means to call the delete function on each address. The Address will be marked as removed and registers itself in the UnitOfWork to be removed. After all addresses have been processed, the person itself can be registered to be removed. Because the *Unit of Work* remembers the correct order, all addresses will be removed before the person is removed. Figure 6.8 shows this procedure. *Lazy Load* is also used to make sure that all objects are up to date but it is omitted in the sequence diagram.

6.8 Attributes

Here is a list of all custom attributes which *STORM* defines:

Table 6.1: List of all custom attributes in *STORM*

Adder	ParameterDef
Column	PrimaryKey

DomainObjectImpl	Table
Factory	ToMany
Finder	ToOne
GenerateCode	VersionField
MapperImpl	

The following description list describes each attribute. This is only a short description meant as an overview. Parameter description is omitted.

Adder Defines a method to add any objects to the current object. For example, it can be used to add an `Address` object to a `Person` object. This attribute is usually used for a ToMany relation. It is needed because an adder could not be called it it was generated automatically in the domain object implementation.

Column Defines the mapping between a member variable and a column in the database. It is used for properties only. Out of these properties, the corresponding member variables are generated.

DomainObjectImpl This is an internal only attribute. It must not be declared in an abstract class. Instead, it is automatically declared in the generated code and is used by the registry to know which generated classes are to be treated as a domain object implementation and should therefore be registered in the registry.

Factory This attribute is used to specify the factory. It is used to attribute an abstract class declaration. Within this class, create methods can be specified. Out of this specification, constructors are generated which can be called from within a user written client program. A class specified with the Factory attribute must be enclosed by the main abstract class.

Finder Like the Factory attribute, this is an attribute which is used for an abstract class. Within this class, abstract methods can be declared. Out of these declaration, custom defined finder methods are generated.

GenerateCode This attribute is used for every user defined, abstract class to indicate that code for this class should be generated. If omitted, the class will be completely ignored by *STORM*.

MapperImpl This is an internal only attribute. It must not be declared in an abstract class. Instead, it is automatically declared in the generated code and is used by the registry to know which generated classes are to be treated as a mapper implementation and should therefore be registered in the registry.

ParameterDef This attribute is used in within the abstract Factory and Finder classes. They define the mapping between parameters and the corresponding properties.

PrimaryKey Declares that the primary key in the database is represented by this property. This attribute must always be used in conjunction with a Column attribute.

Table This attribute maps the whole class to a table in the database. At this time, one class can only map to exactly one table. Additionally, this attribute declares if the PrimaryKey is a surrogate key or a compound key.

ToMany Declares a relation to a collection of other objects. This maps a 1:n relation in the database to in-memory objects.

ToOne Declares a relation to another object. This maps a 1:1 relation in the database to in-memory objects. It must be used to specify the back link to a ToMany relation. For example, if a `Person` specifies a ToMany relation to `Address`, `Address` must declare a ToOne relation back to `Person`. This attribute is always used in conjunction with a Column attribute.

VersionField Specifies a version field in the database. This field is a auto-generated field of the data type `timestamp` and is used to check if the in-memory object is in sync with the database object.

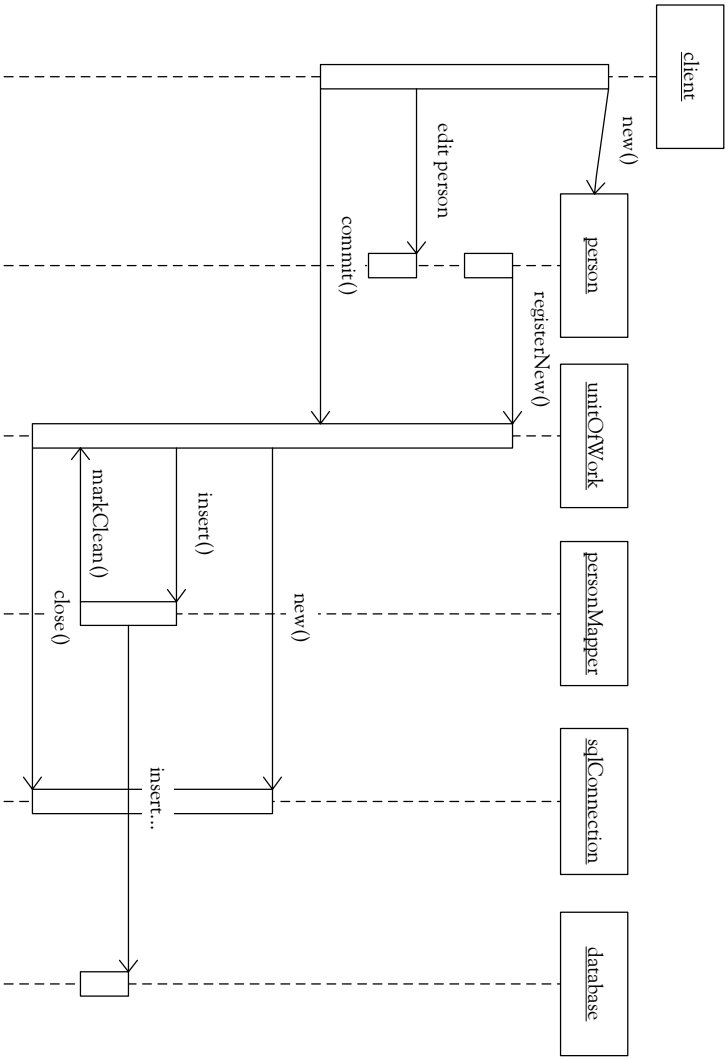


Figure 6.6: Creating a new Object

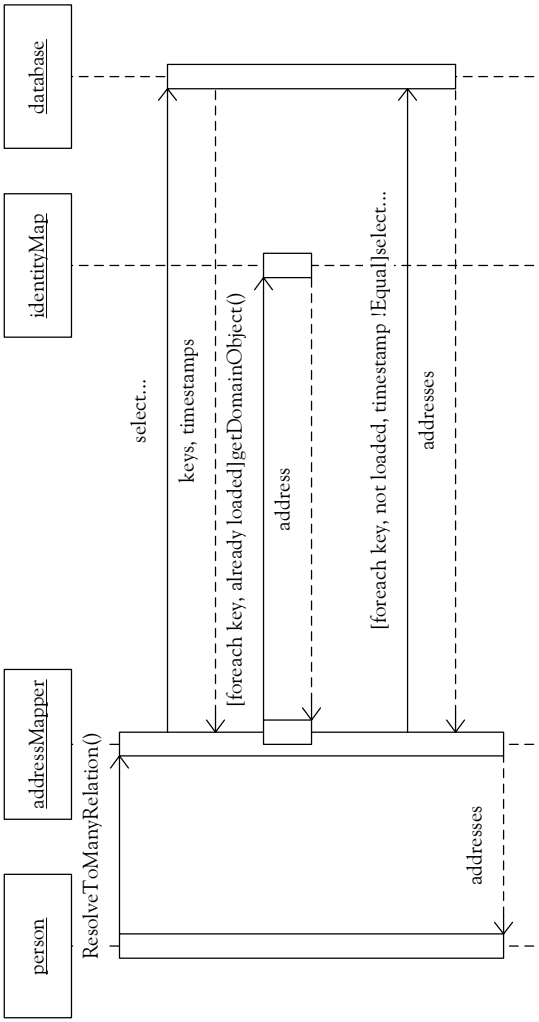


Figure 6.7: Lazy Load mechanism

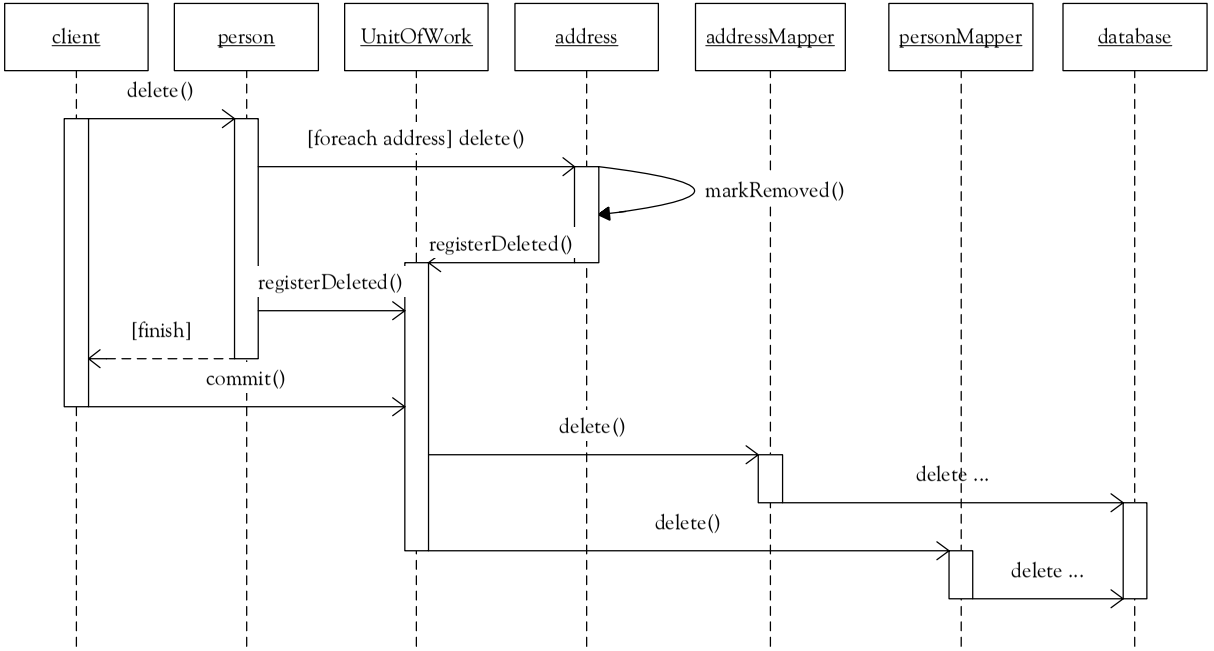


Figure 6.8: Delete an Object

Chapter 7

Creating an Application

7.1 Introduction

The previous chapters were all about *STORM*. This chapter should give you an overview of *how to use* it. Even though it would be possible to create a new application in a relatively short time, we stay with our order application example. This is not meant as a step by step guidance but as a "not so short introduction to create an application with *STORM*". Some parts in this chapter have already been mentioned. They are repeated to put them in the context of creating a new application.

7.2 Starting a Design

In the case of the order application, the database design was already existing. Therefore, this has been taken as starting point. Usually, the starting point would be a domain model but it makes not much of a difference. The database design is shown in Figure 7.1. The corresponding conceptional domain model is shown in Figure 7.2. The domain model looks essentially the same as the database model. This is because every domain object will be mapped to a table in the database. The design of an application is always the starting point and should be done carefully.

Our order application is not very difficult to explain. It contains persons with addresses. One person can have multiple addresses. But an address

in turn belongs always to only one person. Furthermore, a person can have multiple orders. An order contains order details which describe the order and its details like price and quantity. A product then is related to an order detail.

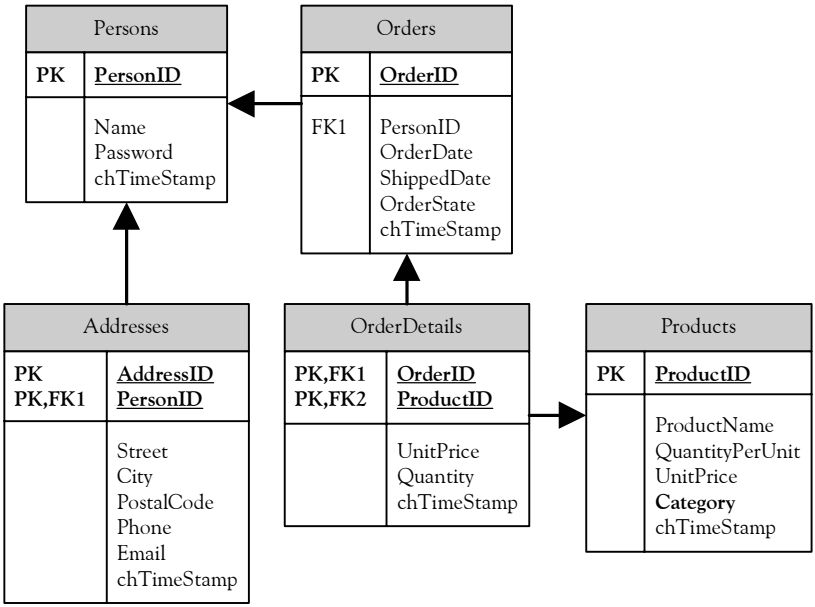


Figure 7.1: Database Design

7.3 Preparation

In order to work with *STORM*, some preliminary work needs to be done. Namely that is to make the ImplGen and MapperGen templates available for the current project. This means, they must exist on the file system. Where to put them is the decision of the developer. The templates can be references by providing the full path. We decided to place the templates in a folder in the current project. The next step is to add a reference to the new project.

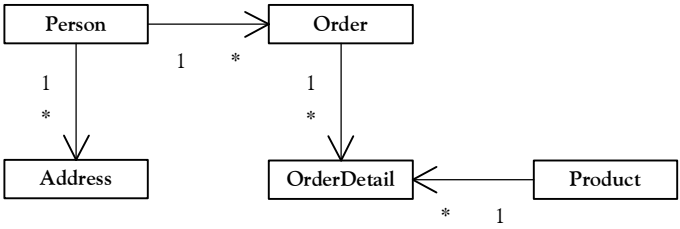


Figure 7.2: Conceptional Domain Model

This reference must point to the *STORM* dll file. And last, CodeSmith [1] must be installed. Thats it.

7.4 Writing abstract Classes

When all the design and preparation has been finished, it is time for *STORM* to come into play. The first step is to write an abstract classes for each class of the domain model. In our case this means to write an abstract class for `Person`, `Address`, `Order`, `OrderDetail` and `Product`. It is very important to write these classes carefully. Listing 7.1 shows the abstract class `OrderDetail`.

Listing 7.1: Defining OrderDetail as abstract class.

```
[Table("Orders", true),
VersionField("chTimestamp"),
GenerateCode]
public abstract class Order : DomainObject
{
  [Factory]
  public abstract class OrderFactory
  {
    public abstract Order createOrder(
      [ParameterDef("Person")] Person person,
      [ParameterDef("OrderDate")] DateTime orderDate,
      [ParameterDef("ShippedDate")] DateTime shippedDate,
      [ParameterDef("OrderState")] String orderState);
  }

  [Finder]
  public abstract class OrderFinder
```

[1] http://www.ericjsmith.net/codesmith/

```
18    {
        public abstract IList findByOrderDate(
20        [ParameterDef("OrderDate")] DateTime orderDate);
        public IList findOrderedBetween(DateTime startDate, DateTime endDate)
22      {
          return new ArrayList();
24      }
        public IList findShippedBetween(DateTime startDate, DateTime endDate)
26      {
          return new ArrayList();
28      }
      }
30
      [Column("OrderID"),
32    PrimaryKey]
      public abstract int OrderId {get;}
34
      [Column("PersonID"),
36    ToOne(typeof(Person), "PersonId")]
      public abstract Person Person {get; set;}
38
      [Column("OrderDate")]
40    public abstract DateTime OrderDate {get; set;}

42    [Column("ShippedDate")]
      public abstract DateTime ShippedDate {get; set;}
44
      [Column("OrderState")]
46    public abstract string OrderState {get; set;}

48    [ToMany(typeof(OrderDetail), "Order")]
      public abstract IList OrderDetails {get;}
50
      [Adder("OrderDetails", "Order")]
52    public abstract void addOrderDetail(OrderDetail od);

54  }
```

All other abstract classes must be defined analogical to this one. Additional methods can also be defined in this abstract class. They are defined without custom attributes. Such methods are ignored by the process of code generation.

If all abstract classes are implemented, the main work of writing the core application is done.

7.5 Configuration Files

The next step is to generate all domain and mapper implementations. This is done using CodeSmith's Custom Tool. In order to use CodeSmith from

within the Visual Studio .NET, an XML configuration file must exist. In this configuration file are all information that CodeSmith needs to generate the code. It would be possible to have just two XML files per project, one for the ImplGen and one for the MapperGen template. This would generate all domain implementation code in one file and all mapper code in another. We decided to split the code and make two files for each abstract class. A Sample of such an XML file is given in Listing 7.2. That is an XML configuration file for an `Order` class. Out of this, the domain object implementation will be generated. The same file is needed for the mapper implementation. The only difference is that the template would not be ImplGen but MapperGen (see line 24). Depending on the application, also different namespace imports would be needed. The generated file has the same name as the XML file, e.g. if we name the XML file `OrderImpl.xml`, the output file name would be `OrderImpl.cs`.

Listing 7.2: CodeSmith Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<codeSmith>
<namespace>HsrOrderApp.BusinessLayer.DomainModelImpl</namespace>
  <imports>
    <import namespace="System" />
    <import namespace="System.Collections" />
    <import namespace="System.Reflection" />
    <import namespace="Storm.Lib" />
    <import namespace="Storm.Attributes" />
    <import namespace="HsrOrderApp.BusinessLayer.DomainModel" />
  </imports>
  <propertySets>
    <propertySet>
      <property name="assemblyLoader">
        <AssemblyLoader>
          <SourceAssembly>
            C:\HsrOrderApp\BusinessLayer\bin\Debug\BusinessLayer.dll
          </SourceAssembly>
          <ClassName>Order</ClassName>
        </AssemblyLoader>
      </property>
    </propertySet>
  </propertySets>
  <template path="..\Templates\ImplGen.cst" />
</codeSmith>
```

7.6 Generating the Code

There are two ways to generate code without the CodeSmith GUI. You can choose between a command line tool and the possibility to integrate the generation into Microsoft Visual Studio .NET. Both take the settings from an xml file as described in the last section.

The command line tool is very useful for creating makefiles. The integrated tool is easy to use while writing code. Both methods can be used and both work.

To run the generation of code from within the Visual Studio you have to add the xml configuration file to the Visual Studio project. Then you have to set the property `Custom Tool` to `CodeSmithGenerator` as shown in Figure 7.3 on the right.

The `Build Action` field has to be set to `Content`. This causes Visual Studio to execute the generation every time the content of the xml file has changed. The generation of the code can also be initiated by right clicking the xml file and select `Run Custom Tool`.

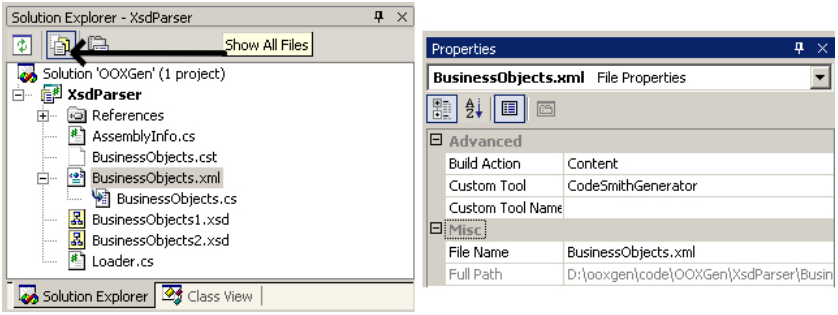


Figure 7.3: Visual Studio .NET integration

If the option *Show All Files* (see Figure 7.3) is turned on, the generated file is displayed as a child of the xml file.

7.7 Working with the Application

Now, everything is created and ready to use. To actually use the code we need a client application which uses the generated classes. This can be of any complexity and size. For this example, we use a very basic console application. Listing 7.3 shows the code of the test method. First, and important, the program class the registry's `init()` method. Parameters are the running program (this) and the host and name of the database to be used (line 3). After initialisation, the application searches all persons in the database. Next, a specific person is searched. That is, the person with the Id 2. All orders for this person are wrote to the console. To finish the sample program, a new order for this person is created. This order is added to the person and changed afterwards.

After everything has been done, a commit on the unit of work is called. This executes all operations on the database.

Listing 7.3: Sample console application

```
public void test()
{
  Registry.Instance.init(this, "localhost", "OrderApplication");

  IList persons = (Person.PersonFinder)Registry.Instance.
    getFinder(typeof(Person)).findAll();

  Person person = (Person)Registry.Instance.
    getFinder(typeof(Person)).findById(new Key(new object[] {2}));

  foreach(Order order in person.Orders)
  {
    Console.WriteLine(order);
  }

  Order order = ((Order.OrderFactory)Registry.Instance.
    getFactory(typeof(Order))).
    createOrder(person, new DateTime(2003, 12, 1), null, "");

  person.addOrder(order);

  order.ShippedDate = DateTime.Today;

  UnitOfWork.Instance.commit();
}
```

Although this is a very simple application, it should give an impression how fast an application can be build using *STORM*. A great advantage of using a generative programming technique is, that even if an application

becomes larger, there is not much extra work to do. The only thing which needs to be done is to implement another abstract class.

Chapter 8

Conclusion

8.1 Specific

A generative programming technique can be very powerful if used in the right context. Even though, it has its limitations, at least where productivity is concerned. One important thing is to precisely define the scope and the target when developing a tool such as *STORM*. That is, it is important to know what the application which should be generated at the end needs to do. For example, to write templates for *STORM* that can be used in *any* project which uses a database is nearly impossible. At least it would lack the whole benefit of code generation. Namely these benefits are that an application can be build in a much shorter time and the resulting application is error-free. But if every possible requirement had to be covered by the template, it would likely take a longer time to develop it than it ever can save.

8.2 In General

In general, a generative programming technique is useful whenever the requirements for a project can be accurately specified from the beginning and do not change very much. Furhtermore, the project needs to be of a certain extend because it is doubtful if its worth the effort to create a framework for code generation just for a small application.

The conclusion therefore is that code generation can help very much to

shorten development time and to improve code quality. This, of course, is not limited to an object/relational mapper.

8.3 Outlook

STORM could be extended in many ways. One important thing would be to implement different locking mechanism, another to support all database mapping types. Another extend would be to generate code for web service facade to domain object mappings. Although *STORM* does not support this, it should give a reasonable start for further development.

Appendix A

Glossary

A

assembly A collection of one or more files that are versioned and deployed as a unit. An assembly is the primary building block of a .NET Framework application. All managed types and resources are contained within an assembly and are marked either as accessible only within the assembly or as accessible from code in other assemblies.

attribute A .NET class used to describe code elements. This attributes can be read by reflection on runtime.

C

CodeSmith A freeware template-based code generator.

D

DLL "Dynamic Link Library" in .NET is an assembly.

G

GAC "Global Assembly Cache" is the place where .NET Assembliesy can be registred. The Assembly needs to be signed. It is possible to register different Versions of the same Assembly.

I

IDE "Integrated Developing Environment" is a tool for developers. Such as Visual Studio .NET.

M

makefile File with a description of a build process. It defines dependencies between several tasks. Nmake is the .NET tool to execute the file.

mapper An object-relational mapper maps the data from an object to a table on the database and vice versa.

P

primary key The primary key of a relational table uniquely identifies each record in the table. It can either be a normal attribute that is guaranteed to be unique or it can be generated by the database.

R

reflection A mechanism to get information about running code.

T

transaction An inseparable list of database operations which must be executed either in its entirety or not at all.

X

XML Schema Definition (XSD) specifies how to formally describe the elements in an XML document. This description can be used to verify that each item of content in a document adheres to the description of the element in which the content is to be placed.

Bibliography

[1] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003. ISBN 0-321-12742-0. 560 pp.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, 1997. ISBN 0-201-63361-2. 416 pp.

[3] Microsoft. Microsoft patterns and best practices. URL http://msdn.microsoft.com/architecture/patterns/MSpatterns/.