UNIVERSITY OF OSLO
Department of Informatics

Fluenty

A type safe query API

Master thesis

Robert Larsen

Spring 2012

Abstract

Several tools for Object Relational Mapping (ORM) have been developed in an attempt to alleviate the mismatch between object oriented programming languages and relational databases. Queries in such tools is written either with a textbased query language or as program code with an API. With a textbased language, features like tool support, compiletime syntax checking and type safety is absent. Such features is to some extent usually available when using an API, but it often results in complex code and poor readability.

This master thesis presents different ORM-tools and their query writing techniques, before it presents a proof-of-concept prototype of a new query API, Fluenty. When implementing Fluenty, we have identified general patterns and techniques suitable when implementing fluent, DSL-like APIs in Java. These findings are presented and discussed. Finally, Fluenty is evaluated against a range of query writing techniques provided by other tools.

Contents

List of Figures

List of Tables

Listings

Preface

This report is the result of work performed on my master thesis at the Department of Informatics at University of Oslo. The topic was presented by BEKK Consulting A/S. Although the work was initiated by an external company, the results of the work are publicly available.

The thesis has been supervised by Ragnar Normann. I would like to thank him for all his help, advice and valuable guidance during my work.

I would also like to thank Kristoffer Dyrkorn and Eivind Barstad Waaler at BEKK, and Stein Krogdahl at Department of Informatics, for guiding me to relevant litterature.

Last but not least I would like to thank my external supervisor Rune Flobakk at BEKK. His enthusiasm and passion is both engaging and motivating for me. He has supported me during the entire project with assistance with technical challenges and thorough reflections around all my questions and thoughts.

Oslo, 1st February 2012

Robert Larsen

Chapter 1

Introduction

The main goal of this chapter is to present the topic and problems addressed in this thesis. It states the problem formulation and clarifies this by providing motivational and background information. Finally, it presents an outline of the report structure.

1.1 Background

Even if object-oriented and object-relational database systems have found their way into the market, relational databases are still dominant. There is a mismatch between the object-oriented model used in modern programming languages like Java, where data are stored in objects, and the relational model used in relational databases, where data are stored in tables. The mismatch is both between the object model and the relational model as well as between the object-oriented programming languages and the relational query languages. This is often refered to as the *impedance mismatch*[6] (cited by [30]).

The term impedance mismatch is broad, and can be refined into more specific problems. Some of the problems pointed out in [16] is explained below.

1. **Instance.** An object is an instance of a class, and might have an arbitrary structure. According to [16], a "row is a statement of truth about some universe of discourse". But, how should an object and a row correspond to eachother? How are the object's states maintained? How much information is needed?

2. **Structure.** A class may be part of a class hierarchy. Its structure and semantics (defined through methods) might be arbitrary. How can such structures be represented in relational database tables?

3. **Encapsulation.** An object's state is modified by methods accessing the fields of that object. A row's state however, has no such protection and may be modified by other applcations. How to ensure data consistency between the object and a row?

4. **Ownership.** A class model is commonly owned and maintained by a team of software developers, while a relational schema often is owned and maintained by a database team. It may be used by other applications and hold legacy data. How to maintain the necessary consistency and correspondance between the class model and the relational schema?

One way to store objects in a relational database is to iterate through the objects, retrieve the values of each object's attributes, and save them into an appropriate table. The values must be stored in such a way that it will be possible to reverse this process, i.e. to read the objects back into the object-oriented data structure. Each value has to be retrieved from the correct table, *new* objects have to be created, and each object's attributes have to be assigned the correct values just retrieved.

With this approach, the developer is responsible for maintaining two parallel data structures, both in the program logic and in the database[2]. This might be a feasible task when developing small applications, with a limited amount of different objects and structures. However, this is not a trivial task in larger applications.

1.2 Object-Relational Mapping

Object Relational Mapping (ORM) is a technique that tries to alleviate this mismatch, and is an alternative to the more cumbersome approach just described.

Several ORM-tools are available today. They aim to make working with object-oriented languages together with relational databases more easy and efficient. Tools like Hibernate and Squeryl are discussed in chapter 2. Additional tools are TopLink and OpenJPA, commonly used with Java and related technologies, Active Record with Ruby, and GORM for Groovy[27]. NHibernate and LINQ are popular tools among many .NET-developers.

ORM-tools add a new layer to the application, between the business logic layer and the data layer. They act as intermediaries between the database and the program code[30]. Instead of accessing the data layer directly, queries are instead run against the new ORM-tool layer, which takes care of the communication with the data layer, invisible for the developer.

1.3 Problem formulation

Queries are commonly written by using one of two techniques. They can be written with an API, using the programming language (e.g. Java), or as plain text using a string based query language. Several tools have their own query languages, usually closely related to SQL. Such query languages are often used to write complex queries, where the API might not provide the necessary expressiveness.

Many developers are familiar with SQL. It often requires a steep learning curve to acquire the knowledge necessary to work efficiently with an API. It might be necessary to invest time to learn and master it properly. Developers might feel that an API reduces their expressiveness, and that they can achieve the results they want easier and in less time with SQL. The API might be felt as a constraint. Additionally, code complexity often increases significantly, which leads to poor readability and understanding.

This means that an API often is not used at all, resulting in queries using the tool's query language instead. This has several drawbacks (Some applies when using APIs as well. Such situations will be discussed later):

- No checking of query syntax at compiletime
- No tool support (e.g. auto-complete and refactoring functionality in an IDE[1])
- Lack of type safety
- Possible to query for non-existing object types
- Reusability is difficult
- Poor readability

This thesis will try to address these issues with the following problem formulation:

1. Create a new proof-of-concept prototype of a type safe query API, called Fluenty
2. Identify typical constructs and patterns used during the development of the prototype
3. Evaluate and compare the prototype's strengths and weaknesses against existing techniques for writing and executing queries

The prototype will be referred to as Fluenty later in this report. The "fluent"-part of the name originates from the term "fluent interface", which will be discussed later in this report. In short, it means more readable and understandable program code syntax with a flow more similar to natural language than traditional syntax. Queries in Fluenty are written following this principle.

This project is not intended to result in a tool ready for professional use. It should rather serve as basis for a potentially more complete future implementation. The main goals of implementing Fluenty can be summarized as follows:

- Explore methods and constructs, and identify patterns for creating DSL-like APIs in Java (a discussion of DSLs and APIs is presented in section 1.7)

[1] IDE = Integrated Development Environment, e.g. Eclipse or NetBeans

- Show how these findings can be used to create a DSL-like API, by creating a prototype of a type safe query API, for writing and executing queries against a relational database

1.4 Project initiator

The project was initiated by BEKK Consulting A/S, referred to as 'BEKK' later in this report. BEKK is a Norwegian business and technology consulting company [2]. Many of their customers are large enterprises and agencies within the public sector.

It should be emphasized that BEKK does not want to keep any results or findings from this thesis exclusively for in-house work. The produced software will be publicly available.

1.5 Method

Implementation has gained much focus in this project - to create a functional prototype with enough functionality so that it can be used to solve simple, but realistic, tasks. Fluenty only provide basic functionality for writing and running queryes. However, the current version of Fluenty serves the purpose of this project - proof-of-concept. It has been strived to develop it in such a way that it will be easy to extend with new functionality later.

More important than Fluenty's functionality is how we can use experiences from the development to identify typical constructs and patterns appropriate when creating fluent, DSL-like APIs in Java (in relation to point two in the problem formulation presented eaerlier).

The idea and desire to work with a technical focus came up when reading about the term "constructive research", described in [17]. They state that the creation of artifacts contributing to a discipline is a common task of research in information systems. However, it is claimed in [9] that the software engineering discipline in some respects still can be considered as immature due to:

- software systems are delivered with a considerable amount of errors (bugs),
- projects are commonly not delivered on-time,
- implementations commonly do not satisfactory meet the clients' requirements.

In most industries, these kinds of failures would not be tolerable at all, but *"(...) software technology is in the Stone Age. Application developers, managers, and end users are paying the price"* [17]. Software engineering needs extensive constructive research to positively support the development

[2] http://www.bekk.no/English/

of the discipline itself. Frameworks, automation of routine work, and formalized processes are examples of such contributing constructs [17].

1.5.1 Development research

To try to contribute with such a construct, this project has been conducted after the principles of development research. The approach has been described by e.g. Villiers[7] and van Den Akker [29]. Development research aims to contribute in both theoretical and practical ways. Another related approach, probably more well-known, is action research. As pointed out in [7], action research does not always produce new solutions. And, if it does, the solution can not always be generalized. This thesis aims to provide both a new solution and a solution of which at least parts of it can be generalized. This is espescially important considering that identification of typical constructs and patterns is an important part of this project. Therefore, development research was considered as a more suitable approach.

According to [7], development research has a dual focus:

1. develop practical and innovative ways of solving real problems

2. propose general design principles to influence future decisions

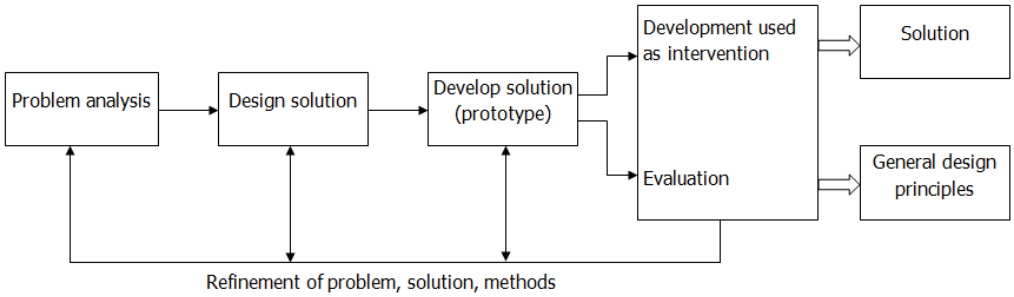

Figure 1.1: Development research

Figure 1.1 summarizes the most important phases of the work.

1.5.2 Academic resources

Many of the tools mentioned in this report have not been subject for academic study. The availability of sources holding the sufficient level of reliability has often been limited. Much of the material exists in informal forms, such as postings in different technical blogs or message boards. Where it has been impossible to find other sources, a URL is provided as a footnote. More formal resources are used as references.

1.6 Research questions

The development of Fluenty represents this project's practical contribution. Its theoretical contribution will be made by trying to answer the following research questions:

1. **What are the main differences, in terms of expressiveness, between Fluenty and other ORM query techniques?** Queries will be compared on the basis of:
    - Preparations
    - Query writing
    - Functionality
    - Understandability
    - Result handling
2. **Which constructs can be identified as typical for this type of API-design?**
3. **How to create an architecture well suited to be integrated with other existing ORM-tools?**
4. **In what other areas than this project might proxy objects be a useful contribution?**
5. **How well did the selected research and development methods suit the project?**

1.7 API, Framework or DSL?

When reading about the topic in different books, reports, message boards and blogs, terms like "API", "framework", "library" and "DSL" often seem to be used randomly and interchangeable. To avoid that in this report, this section will list and clearify the definitions and state what they refer to in this report.

API is an abbreviation for Application Programming Interface. In object-oriented languages, this is a set of class defintions. Each of the class definitions has a set of behaviours associated with them. A behaviour is a rule for how an object acts in given situations. Wikipedia[3] provides the following defintion: *"..is a particular set of rules and specifications that software programs can follow to communicate with eachother"*. An implementation of an API is usually called a **library**. A library provides functionality that can be re-used by developers. The probably most well known library is the Java API, also called the Java Class Library.

This thesis does not make a clear distinction between API and library. Every mentioned API has at least one corresponding library implementation, even if it is beeing referred to as an API.

[3] http://en.wikipedia.org/wiki/Application_programming_interface

A **Framework** is a collection of libraries. In addition, some key features separates them from libraries[4]:

- Framework program code can not be modified by the user. However, it can usually be extended. This is typically done by providing a new implementation of certain methods (overriding).
- The flow is not controlled or dictated by the user, it is handled by the framework (inversion of control).

DSL is an abbrevation for Domain Specific Language. It is a programming- or specification language tailormade for a specific problem domain. Fowler [11] differs between external and internal DSLs. An external DSL is written in a language different from the programming language used in the application. An example is Hibernate (2.1) configuration files, written with XML. An internal DSL uses the same language as the application, but only a subset of its features.

Some of the tools mentioned in this report are referred to as DSLs. The difference between an internal DSL and an API might seem vague. Fowler [11] means that the difference lies in the language nature. In an API, each method's name should make sense on its own. The methods of an internal DSL often only make sense in the context of a larger expression in the DSL.

In section 1.3, we introduced the term "fluent interface" briefly. With such syntax, methods are often designed to be part of a longer chain. That is, they are given names that form a natural "sentence" when the methods are chained together. According to Fowler's definition, a fluent API (an API with a fluent interface syntax) can therefore be considered as an internal DSL. This is what we mean when we refer to "DSL-like APIs" in this report.

We have chosen to not make a clear distinction between DSL and API. Some might probably consider Fluenty as a DSL while others will refer to it as an API.

1.8 Report structure

This report is organized into 8 chapters.

Chapter 1 describes the context, background, problem domain and motivation for this project. It states the problem formulation and provides a description of the research questions and methods used during the project, as well as a short readers guide.

Chapter 2 provides more background informatiom by giving an overview of existing solutions.

Chapter 3 presents Fluenty from a user's point of view, with focus on its functionality.

Chapter 4 presents Fluenty from a more technical perspective by describing the different tools and frameworks used in the development.

Chapter 5 contains a discussion of Fluenty's design and identified patterns.

[4] http://en.wikipedia.org/wiki/Software_framework

Chapter 6 presents a discussion of findings regarding the research questions presented in chapter 1.

Chapter 7 provides an evaluation of Fluenty. The focus is on to what extent it meets the requirements specified in chapter 3.

Chapter 8 presents a summary of the work, as well as pointing at different aspects that might be looked into in future work.

Chapter 2

Existing solutions

It does exist solutions that in different ways address the impedance mismatch. The following subsections provide an introduction to some of them.

Hibernate will be described most thorougly. This is a very popular framework today. Hibernate is a very comprehensive framework. It provides a nearly complete solution for all aspects in an application that uses ORM. Working with Hibernate has helped me to understand the problem domain. Therefore, the description of Hibernate in the next subsection is provided as background information.

2.1 Hibernate

Hibernate is a tool for *"automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using meta-data that describes the mapping between the objects and the database. ORM, in essence, works by (reversibly) transforming data from one representation to another"*[4] (cited in [30]). According to the official Hibernate website[1], Hibernate is a "collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping."

POJO is an abbreviation for "Plain Old Java Object". It is a simple, ordinary Java object. It does not extend or implement any frameworks or interfaces from other libraries and APIs, and is bound to the rules defined in the Java Language Specification only.

Queries can be written with several techniques, like Hibernate Query Language (HQL), Hibernate Criteria API, JPQL[2], and native SQL. It is assumed that the reader is familiar with SQL, and since JPQL and SQL are closely related, these two will not be covered in this report. HQL and Criteria will be covered in subsection 2.1.1 and 2.1.2 respectively.

[1] http://www.hibernate.org

[2] Java Persistence Query Language

2.1.1 Hibernate Query Language

Hibernate Query Language (HQL) is Hibernate's native query language. Syntactically, HQL has much in common with SQL and different SQL dialects. However, it is a fully object-oriented language. It provides functionality for expressing inheritance, polymorphism and associations [19].

HQL-queries are written as strings. Similar to SQL, queries are written on the form "select from where". However, the select-clause is not mandatory. The simple query `FROM Book` is equivalent to `SELECT * FROM books` in SQL.

A major area where HQL differs from SQL is that HQL-queries are written against persistent objects (entities) in the application (e.g. objects of class `Book`), rather than against the database tables (e.g. books). This means that the developer does not need to be familiar with the database schema; he can work with the same domain objects as in the program code. Additionally, this means that instead of returning just fields from database tables, HQL can retrieve and return persistent objects directly.

HQL offers possibilities for associations, joins, aggregations, operators, order by and group by. Queries are case-insensitive, with the exception of class names and properties.

HQL is, as mentioned, an object-oriented language, and it fully supports polymorphism. Hence, if the class `ProgrammingBook` is a sub-class of class `Book`, an HQL-query which asks for all objects of type `Book` will also return all objects of type `ProgrammingBook`.

HQL-queries are not checked by the compiler. The compiler will thus not complain if the developer writes "Books" instead of "Book", i.e. an object that does not exist. If a query returns objects of type `Book`, while objects of a completely different type, like `Car`, were expected in the code, it will cause a runtime error.

```
1  private List getMyObjects()
2  {
3      SessionFactory f = HibernateUtil.getSessionFactory();
4      Session session = f.getCurrentSession();
5      session.beginTransaction();
6
7      List result = session.createQuery( "from Book" ).list();
8      session.getTransaction().commit();
9
10     return result;
11 }
```

Listing 2.1: Simple HQL-query, retrieving all objects of a certain type

The method in listing 2.1 starts with retrieving the current session. This is not important for this example. Refer to the documentation of HQL [3] for more information. A more important aspect is the query, on line 7. The query in this example is the simplest query possible; return all objects

[3] http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html

without any restrictions. Note the call to the method `list()` at the end of the line. This method returns the results from the query as the Java-type `List`. However, it is not parameterized. It is not possible to determine at compiletime what type of objects the query will return. Hence it is not possible to express that the list should only contain objects of type `Book`, and no type safety is attained.

2.1.2 Hibernate Criteria

Hibernate Criteria API, referred to as Criteria in this report, is an API for writing queries as Java program code, and not as strings. This is the main difference between Criteria and HQL, as they offer almost identical functionality[23]. Criteria supports sorting, assosiations, and aggregations just like HQL.

It is claimed that Criteria is the easiest way to retrieve data[23]. This claim is subject for discussion, and when to choose which of the two is discussed in 2.1.3.

```
1 private List getMyBooks()
2 {
3     Criteria crit = session.createCriteria(Book.class);
4     crit.add(Restrictions.gt("price",new Double(99.0)));
5     crit.add(Restrictions.like("name","K%"));
6     List list = crit.list();
7
8     return list;
9 }
```

Listing 2.2: Simple query written with Criteria

On line 3, there is an important thing to notice. Instead of typing the object-type with a string, the object-type is set with Java code, i.e. with `Book.class`. To make this code compileable, the class Book needs to exist. It is therefore impossible to execute a query that asks for objects of a non-existing type.

On line 4 and 5, two `Restrictions` are added. When adding these, only objects having the property "price" set to a value higher than 99 and the property "name" set to a value starting with the letter K will be returned. Note that these are specified as strings. Typing errors and type incompatibilities remain undetected until runtime.

Like HQL, Criteria is fully object-oriented and supports polymorphism.

2.1.3 Criteria or HQL?

As long as HQL and Criteria are so similar in functionality, it will depend much on the knowledge and experience of the developer which of them that is preferable. If the developer has good knowledge of SQL, then HQL might be the right choice. The differences between them are few. If the developer is an experienced Java-developer, he might feel more familiar

with Criteria, as this technique involves more writing of Java-code than HQL does.

In addition to the developer's preferences, the type of application or task also has an impact. *"Hibernate Criteria is without doubt tailor-made for dynamic query generation" [28].* Dynamic queries are queries that are generated on the fly, i.e. based on user input. Examples are applications for ticket reservations and applications with complex search-functionalities. In such applications, it is impossible to know at the development stage what the contents of each query will be, as this will depend on what the user gives as input. For such cases, Criteria will usually be a better choice than HQL. Because a HQL query is a string, it will involve string manipulation and concatenation to get the correct user values into the correct places in the query. It is easier to just pass each value as a parameter to the query with Criteria. On the other hand, if the query is static, i.e. it will not change depending on user input, HQL will probably be a better choice. Using Criteria here will generate more complex code which is harder to maintain than an HQL query[28].

2.2 QueryDSL

QueryDSL[4] is an open-source framework with essentially the same purpose as Fluenty. It was originally developed because of the need for maintaining HQL queries in a type-safe manner, but has evolved since then.

The framework needs to be used on top of an underlying framework. It offers support for writing and executing queries against already stored data only. Mapping and maintenance of the data must be handled by the backend (underlying framework). Currently, Collections, JDO, JPA, JDBC, Lucene, Hibernate, MongoDB and RefBean are supported.

QueryDSL is developed by Mysema[5], a company offering design, implementation and consulting services. QueryDSL is released under the Lesser General Public Licence. Thus, the source code can be downloaded, changed and used free of charge. The project is currently active, with several new releases of the framework each year. It seems to be a relatively active community behind, with a popular discussion board. This, together with comprehensive documentation, should help keeping a low threshold for developers to start using the framework.

2.3 Squeryl

Squeryl[6] is a DSL for the Scala programming language. However, since Scala programs runs on the Java Virtual Machine, we include a brief description of Squeryl here.

[4] http://www.querydsl.com
[5] http://www.mysema.com/en/
[6] http://www.squeryl.org

Squeryl offers functionality for writing and running queries as well as for defining schemas/mapping, transaction handling, data persistence and maintenance. Thus, it is not dependent of having an additional framework as backend. It can be used together with most well-known database systems, and supports Postgres, Oracle, MySQL, H2, DB2, MSSQL, and Derby.

There is no commercial company behind Squeryl[7]. It started as a hobby project by a single developer. Today, it's beeing developed by the founder and a dozen other contributors. Contributions to Squeryl seem to be done several times per month, via its GitHub repository[8]. Squeryl is released under the Apache 2.0 Licence, allowing the source code to be downloaded, changed and used free of charge.

An active community provides users of Squeryl with support. A dedicated discussion board serves as the primary communication channel for this. Quite comprehensive user guides are available on the project website, as well as on several other discussion boards and blogs. The availability of reliable documentation has increased noticeably during the work with this project, and Squeryl seems to have gained more popularity lately.

```
1 class Book(val id: Long, val name:String) {
2
3   def authors = from(BookDB.authors)(a => where(s.authorId === id)
        select(a))
4
5 }
```

Listing 2.3: Query with Squeryl

2.4 JPA 2.0

The Java Persistence API is included in the Java platform. It cannot be considered as a "tool" like the ones described so far. It's just a specification, a set of interfaces that require implementation [18]. Version 1.0 was released on 11 May 2006. The current version, 2.0, was released in December 2009.

JPA is not limited to queries. *"The Java Persistence API deals with the way relational data is mapped to Java objects ("persistent entities"), the way that these objects are stored in a relational database so that they can be accessed at a later time, and the continued existence of an entity's state even after the application that uses it ends. In addition to simplifying the entity persistence model, the Java Persistence API standardizes object-relational mapping"*[5].

There are multiple implementations of JPA available, like Hibernate, EclipseLink, TopLink and OpenJPA. JPA 2.0 consists of the specification itself, Java Persistence Query Language (JPQL), Java Persistence Criteria API, and object relational mapping metadata.

[7] http://nikolajlindberg.blogspot.com/2010/09/interview-with-maxime-levesque-author.html

[8] https://github.com/max-l/Squeryl

JPQL is JPA's text-based query language. It is very similar to HQL, described in subsection 2.1.1, as it resembles SQL in syntax and is written against persistent objects in the application rather than against database tables.

JPA 2.0 is contained in the package `javax.persistence`. It contains the Criteria API and JPA Annotations (for object relational metadata). JPA 2.0's Criteria API resembles Hibernate Criteria, described in subsection 2.1.2. It contains methods for writing queries as part of the program code (thus, checked by the compiler). Fluenty uses JPA 2.0 Criteria API to build queries internally. This is discussed more thoroughly in subsection 4.3.6.

2.4.1 JPA Annotations

JPA 2.0 defines several annotations. An annotation is metadata added to the source code. It does not affect the program semantics, but it can be interpreted by different tools and libraries. It is recognized by beeing prefixed with an @.

Each class describing a persistent entity is annotated with `@Entity` before the class definition. The annotation declares this class as an entity, and the class is then mapped to a database table. By default, the table gets the same name as the entity, but if it for some reason is required to have a different name, this can be specified by the annotation `@Table`.

Each entity must have a public or protected constructor with zero arguments. No instance variables can be declared public - each field should be accessed only by the corresponding set- and get-methods. This is according to the JavaBean standard, discussed more thoroughly in section 5.1.

Each entity needs a field containing an identifier, of type long, called `id`, as shown in listing 2.4.

```
1    @Id
2    @GeneratedValue(generator="increment")
3    @GenericGenerator(name="increment", strategy = "increment")
4    private Long id;
```

Listing 2.4: JPA 2.0 annotations

Only `@Id` is mandatory. However, in order to make the id increment automatically the other annotations are also needed.

Fields of a class-type, except String, needs to be annotated with a relationship type like `@OneToOne` or `@ManyToOne` etc.

The selection of available JPA 2.0 annotations is large. Here, only those used during the development of Fluenty are mentioned briefly. A more thorough guide can be found in the Hibernate documentation [9]. It includes e.g. locking strategies, inheritance mapping, and more comprehensive definitions of primary- and foreign keys than just by the `@Id` annotation.

[9] http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html

Chapter 3

Fluenty — requirements and usage

This section presents the functionality of the new API, Fluenty. It describes the requirements and provides a more thorough presentation of relevant central concepts.

3.1 Requirements

This section elaborates the specified requirements, both functional and non-functional. The requirements were not specified as absolute or definite, as the exploration of methods, constructs and patterns was considered as more important than concrete functionality. Hence, the requirements were specified in order to have something tangible as basis for the work rather than as a formalized requirement specification.

Fluenty is designed to be used when developing applications with Java. It offers the developer a type safe, fluent way to write queries against data stored in a relational database.

3.1.1 Functional requirements

The functional requirements, listed below, states the functionality Fluenty must provide.

1. **Retrieve all objects of a specific type**
   It must be impossible to compile the query if it asks for a non-existing type.

2. **Retrieve objects of a certain type, with one or more constraints**
   Adding constraints must be done using the relevant get-methods in the persistent objects. It must be possible to check at least if a value is equal, greater than, greater than or equal, less than, less than or equal. Adding constraints must be done in a type safe manner. That is, it must be impossible to compare incompatible or non-existing values.

3. **Retrieve objects of a certain type, having a collection containing another particular object**
   As an example, if an object of type `Author` is given, retrieve all `Books` with a collection containing that object. The type safety requirements still apply (as for the previous requirement).

4. **Retrieve objects of a certain type, holding a reference to another object, which satisfies one or more constraints**
   Retrieve all `Books` having a reference to an `Editor` with a certain name. Again, the same type safety issues apply. Ideally, it should not be any limits on the length of these chains, we want to be able to write `getPublisher().getEditor().getName()...etc`. However, the minimum is two get-methods chained together. Additional options is considered as a bonus.

5. **Compatible return type**
   A query must return either a single object, or a collection, of the same type as specified in the query. Any type-conversion or similar should be unnecessary, and it should be possible to assign the return value from the query directly to an object variable or a collection.

6. **Return a collection or a single result**
   Query results should be returned as either a collection or as a single object.

These requirements were specified and discussed using user stories. The users stories is presented in appendix A on page 79.

A more thorough description on how the functional requirements translate into practical usage and concrete examples is provided in section 3.3.

3.1.2 Non-functional requirements

Non-functional requirements are requirements that does not directly affect the software's functionality. However, they can still have a large impact on the software architecture and the way the software is developed. According to [3], non-functional requirements are normally defined in the following groups: performance, availability, modifiability, security, testability, and usability.

No requirements regarding availability and security has been defined. It is believed, and assumed, that this is handled sufficiently by the underlying technology and the database.

Performance

- Users should not notice any difference in performance when executing queries using Fluenty instead of JPA 2.0 Criteria API directly.

Modifiability

- Developers should easily be able to extend the Fluenty's vocabulary and functionality.
- Facilitate easy integration with other ORM-tools.

Testability

- The project should use test-driven development.

Usability

- Users should receive feedback about errors at compiletime. This includes both syntactical errors in the query and type compatibility issues.
- Users should be able to write queries using native Java language with a fluent syntax (see subsection 3.2.3).
- Fluenty must be usable in any environment where JPA 2.0 is used already. Users must be able to swith between either two query techniques at their own discretion.

3.2 Central concepts

Until now, terms like type safety and fluent interfaces have just been mentioned briefly when describing Fluenty's functionality. The following subsections provide a more thorough description of these terms.

3.2.1 String-based queries

When using a text-based query langauge, like SQL or HQL, queries are written in the Java program code as plain text. That is, as the type String. Queries are therefore not interpreted and checked by the compiler, and are not compileable as valid program code. A compiler can neither detect any syntactical errors in the query, nor any incompatibilities between datatypes. Thus, tool support for detecting syntactical errors, type compatibility issues and refactoring is missing. That is, the immediate feedback a developer is used to get in his IDE (Integrated Development Environment) about such errors is lacking.

```
1  Query q = entityManager.createQuery(
2  // SQLException is thrown if there is a syntactical
3  // error in the query
4  "SELECT AVG(b.numberofPages) FROM Books b where b.Id = :Id");
5
6  // RuntimeException if id is of the wrong type
7  b.setParameter(1, id); // or if the wrong index is given
8
9  // ClassCastExeption if the result cannot be casted to Number
10 Number avg = (Number) b.getSingleResult();
```

```
11
12 // NullPointerException if the query returns null
13 avg.floatValue();
```

Listing 3.1: Potential runtime errors

Listing 3.1 shows five possible errors that might occur at runtime when using a textbased query language (JPQL). Each of the potential errors could have been detected at compiletime if a compiler had been able to interpret the query. The example is for illustration purposes only and does not necessarily resemble a correct, functional query.

3.2.2 Typesafety

If we go back to the example in section 2.1.2 on page 11, we see that the two properties we add `Restrictions` to, "price" and "name", are given as strings. Therefore, it is impossible for the compiler to check if these properties actually exist. Furthermore, it is also impossible for it to check what kind of datatypes the properties have. We see that on line 4, "price" is compared to a value of type double. For all we know, the type of "price" could be anything other than a double, and the compiler will not complain. At runtime, we probably will get some kind of type-mismatch error, if price has another type. Or, we could get some other type of error if the property "price" does not exist at all.

3.2.3 Fluent interfaces

Queries written with either a textbased query language or an API might be difficuelt to read. One of the intentions for making Fluenty is to make it possible to write queries with a more fluent syntax which lies closer to plain English. This syntax is intended to be more readable and have an easier "flow". Fowler and Evans [10] introduced the term "Fluent interface" for this kind of syntax. Examples of APIs that utilize this technique in an extensive manner are Mockito[1] and LambdaJ[2].

An example can be seen in listing 3.2 (inspired by an example from Johannes Brodwall[3]):

```
1 System.out.println(with(persons)
2     .retain(having(on(Person.class).getAge(), gt(50)))
3     .sort(on(Person.class).getAge())
4     .extract(on(Person.class).getName()));
```

Listing 3.2: Example of a fluent interface

This small piece of Java-code gets all persons from the `Collection` named "persons", with an age greater than 50. Note that the method `getAge()` in the `Person`-object is used to get the age of each person, i.e.

[1] http://www.mockito.org

[2] http://code.google.com/p/lambdaj/

[3] http://johannesbrodwall.com/2010/09/07/dynamic-subclass-apis-make-java-seem-young-again/

the properties are not given as strings. On line 3, the results are sorted, while line 4 gets the name of each person.

A central technique when using fluent interfaces is method chaining, as seen in listing 3.2. Calls to the methods `retain()`, `sort()` and `extract()` are written in one single instruction - they are chained together. As Fowler[11] points out, method chaining and fluent interfaces are not synonymous. Method chaining is only one of several valuable techniques.

The principle is that each method returns an object that can be used for invocations of other methods later in the chain. E.g. `retain()` returns an object containing the method `sort()`.

While method chaining certainly improves readability, it also has some drawbacks. One is that it might make the code harder to debug, as debuggers usually work on a line-by-line, instruction-by-instruction basis [11]. Another drawback is that it might be difficult to see where the chain ends, described by Fowler as the finishing problem [11]. It might be difficult to see if the last method in the chain actually returns an object of the expected type.

3.3 Practical usage

This section presents how Fluenty complies to the the functional requirements, by explaining how it can be used practically.

Writing and running type safe queries using Java, with a fluent syntax, is the core part of Fluenty's functionality. Other database-operations like insert and delete are beyond its scope, since these are very well covered by other ORM-tools.

Fluenty is a proof-of-concept (POC) implementation. It has been developed to prove that it's possible to develop an API for the desired purpose with the utilized techniques. Thus, Fluenty does not satisfy the functional requirements a professional developer has. However, it is designed to be easy to extend with new functionality. Currently, it provides a foundation - a platform to build on.

3.3.1 Retrieve all objects of a certain type

By the example query presented in listing 3.3, it's possible to retreieve all objects of a certain type.

```
1 List<Book> books = repository.find(Book.class, having(on(Book.
    class)).getAll());
```

Listing 3.3: Retreieve all objects of a certain type

The object named `repository` is an object of type `Repository`, and it is the starting point for each query. It contains two methods relevant to mention here, `find()` and `findSingle()`.

The first parameter to both methods is an object type (`Book.class` in the example). `find()` will return a `List` containing elements of that type, while

`findSingle()` will return a single object. `findSingle()` is useful when the user knows that the query will return only one result.

The next parameter can be considered as a constraint. Two central methods in that regard are `having` and `on`. They need to be present in every query. In the previous example, the only constraint is an object type. The examples in the next subsections are better suited to illustrate their purpose.

If the `List` is parameterized with another type than specified by `find`'s first parameter, it will result in a compile error. Thus, full type safety is preserved.

3.3.2 Retrieve objects with constraints

Instead of just retrieving all objects of a certain type, it is possible to return only those that have certain properties, as shown in the example query in listing 3.4.

```
1 Book book = repository.findSingle(Book.class, having(on(Book.class
    ).getTitle()).equal("Some booktitle"));
```

Listing 3.4: Retreieve objects with constraints

This query returns all `Book`s with the title "Some booktitle". It differs from the query presented in the previous subsection in that it uses a get-method, from the class `Book`, and a constraint method. In natural language, this query can be translated into something like "get all books having a title equals to Some booktitle".

We see that `eq` is invoked after the closing parenthesis of `having`, and `having` is therefore necessary to gain access to the constraint methods. Available constraints is listed in table 3.1:

Constraint	**Method name**
equal	`equal()`
greater	`greaterThan()`
greater or equal	`greaterThanOrEqualTo()`
less	`lessThan()`
less or equal	`lessThanOrEqualTo()`

Table 3.1: Currently supported constraints

To avoid compilation errors, the get-method's return type and the type of the constraint method's parameter must be compatible. In the above example, `getTitle()` returns a String. Thus, `eq`'s parameter must also be of type String.

The get-method is invoked after `on`'s enclosing parenthesis. `on`'s main purpose is to make the methods in the persistent object (the `Book`) visible for the user. Methods in the persistent objects can therefore be used directly in the query. That makes it possible for the compiler to verify that the get-method actually exists in that object.

Constraints can be added based on properties in referenced objects. As an example, each `Book` can have a reference to a `Publisher`. If we want to retrieve all `Books` having a `Publisher` with a specific name, this can be expressed with the example query in listing 3.5.

```
1 List<Book> books = repository.find(Book.class, having(on(Book.
      class).getPublisher().getName()).equal("Manning"));
```

Listing 3.5: Retreieve objects with constraints in a chain

`eq`'s parameter must be compatible with the return type of the last get-method in the chain.

3.3.3 Retrieve objects with a certain value in a collection

Sometimes it is not sufficient to add constraints for field values only. Fluenty does therefore make it possible to retrieve objects having a collection of objects which again have a field with a certain value, as shown in the example query in listing 3.6.

```
1 List<Book> books = repository.find(Book.class, having(on(Book.
      class).getAuthors()).having(on(Author.class).getName()).equal(
      "Robert Larsen"));
```

Listing 3.6: Retreieve objects with a certain value in a collections

Like in the previous example, `eq`'s parameter must be compatible with the return type of last get-method in the last chain. Additionally, the parameter to the last `on` (`Author.class`) must be compatible with the return type of the last method in first chain (getAuthors()).

3.3.4 Retrieve objects containing another object

As a supplement to retrieving objects based on field values, it is possible to retrieve objects which have a reference to another object in a collection. If we have an `Author`-object, we want to retrieve all `Books` having that `Author`, as shown in the example query in listing 3.7.

```
1 List<Book> books = repository.find(Book.class, having(on(Book.
      class).getAuthors()).with( author ) );
```

Listing 3.7: Retreieve objects containing another object

The parameter to `with`, `author`, is an `Author`-object. The collection returned by the get-methods must contain objects of a compatible type.

Chapter 4

Implementation

This chapter describes the implementation of Fluenty. It presents relevant methodologies, practices and tools, and how their features have been utilized. It also provides a summary of the most important implementation challenges.

4.1 Methodologies and practices

This section is provided in the context of research question 5 (see chapter 1.6 on page 6), where the selected methodologies will be assesed in terms of their suitability for this particular project. The results of this assessment will be presented in section 6.5 on page 64.

4.1.1 Development methodology

Several different development methodologies, like Waterfall, Unified Process (UP) and Scrum, can be followed when creating an application. Typically, they all involve organizing a team of developers and a customer. In this project, there has been only one developer. Even if the project has an external initiator, BEKK, they can not be considered as a real customer. Real customers will typically demand that the product (the prototype) *must* have *exactly* that functionality and satisfy *exactly* those requirements. Additionally, the aspect that probably is subject for most concern in the majority of real projects, money, has not been involved.

In this project, the essence of the specified requirements has been to acquire more knowledge about the problem area and to try out certain elements in practice. Based on that, and the reasoning mentioned above, it was chosen not to follow one specific methodology, but to utilize only a subset of typical methodology activities.

The usage of user stories is an example of one such activity. They are commonly used in agile methodologies, espescially Scrum, as a tool for requirement identification. In some cases they are used for requirement specification as well. In this project they have been used for both. Since only one developer has been interacting with the "customer" during the work, the chance of misunderstandings and ambiguities was considered

small. Therefore, formalized requirement documents were not seen as a necessity. Spending much time to administrate and maintain a formal specification was considered a waste of time. In that context, user stories turned out to be an informal way to concretize the requirements.

A user story describes that a role (typically a user) wants to use the application to achieve a goal. They usually follow this pattern; *"As a role, I want goal/desire, so that benefit"* [1]. Sometimes they are shortened to just *"As a role, I want goal/desire"*. This pattern has been used in this project.

User stories can be beneficial in that they are formulated with natural language understandable for both customer and developer, providing a better understanding of the application's requirements. They are also short and easy to maintain. The main purpose of using user stories in this project was to identify and specify the requirements. Therefore, more formal parts of handling user stories in Scrum, like backlog and burndown-charts, were found irrelevant and therefore omitted.

4.1.2 Test-Driven Development

"Software tests prove to be the strongest attack in the struggle for high quality, reliable software"[22]. Despite this, testing is often considered to be just extra work, and is postponed until after the implementation is completed. To prevent that from happening in this project, the principles of Test-Driven Development (TDD) was followed.

Two important concepts in TDD is unit testing and refactoring. Unit testing means that each method in each class in the application should typically have a corresponding test method, a unit test. A unit test for a method is written before the actual method is implemented. Of course, the test is then certain to fail. Program code is then written incrementally with refactoring in small steps until the test succeeds [20]. In TDD jargon, the test "goes green".

In this project, JUnit[2] has been used as testing framework. Unit tests is written as regular Java code in a Java class. Such a class is usually called a test suite.

```
1 @Test
2 public void resolveFrom()
3 {
4       Method method = bookClass.getMethod( "getTitle" );
5       Field f = fieldResolver.resolveFrom( bookClass, method  );
6       assertThat( f.getName(), is( "title" ) );
7 }
```

Listing 4.1: A simple JUnit test

Listing 4.1 shows a unit test for the method `resolveFrom()` in the class `FieldResolver`. This example is taken from Fluenty's source code. For now, its purpose and behavior is not important. However, it is described more thoroughly in chapter 5.1.1 on page 39.

[1] Example from Wikipedia: http://en.wikipedia.org/wiki/User_story

[2] http://www.junit.org/

Note that the unit test and the actual method have similar names. A unit test in JUnit is recognized as such because of the annotation `@Test`. Simply put, the purpose of the method `resolveFrom()` is to resolve a field in an object on the basis of a method name. The tested method is invoked on line 5. On line 6 it is verified that it actually returns the expected value. This is done using the method `assertThat()`. The expected value is an object of type `Field` which has a name equal to the string "title". The test is passed if and only if the returned value matches the expected.

The benefits of TDD can be summarized as follows:

- **Incremental development**. The development of code in small incremental steps makes it possible to achieve working software almost immediately. This is a motivational factor, as results can be seen quickly. It might also improve productivity, as the developers' only focus is to make the next test pass.

- **Improved understanding**. No more code than just enough to make the test pass is written [1]. Less amount of code may lead to better overview. No code is developed without a clear purpose - to make the test pass. This makes developers more aware of what each part of the code actually does, and improve their understanding of the software behaviour.

- **Documentation**. A test suite serves as a formal requirement, and it also prooves that the code fullfills that requirement. A passing test proves that the tested unit performs exactly as specified. It may reduce the need for written documentation, as the test itself helps to describe the system behaviour.

Some research has been conducted regarding whether TDD is more effective and efficient than the conventional way of developing code. Among others, Gupta and Jalote [14] has performed some experiments regarding this matter. They found that TDD seemed to reduce overall development efforts and improve the developers' productivity. They also found indications for better code quality when doing TDD instead of conventional approach, but the findings could not be related to TDD with absolute certainty. On the other hand, they found indications that better choices regarding design were taken when using the conventional approach. The participating developers also argued that they felt more confident about the design choices when not using TDD.

Section 6.5 on page 64 presents a discussion of how we have utilized TDD in this particular project.

4.2 Development tools

Various tools have been used during the implementation. This section presents those that will influence any further work.

4.2.1 Maven

When building applications, the need for some sort of project management tool will eventually arise. A developer needs to handle compiling, testing, deployment, resource management, dependency handling, among others. In this project, the tool Maven has been chosen. The main reasons are:

1. Maven facilitates well for TDD
2. BEKK uses Maven in many of their projects. Fluenty can then easily be imported and integrated in existing projects.

On the Maven website[3], Maven is referred to as a "software project management and comprehension tool". It helps with the following aspects:

1. Buildprocess
2. Project structure
3. Dependency handling and management
4. Documentation

Projects is in Maven described using a Project Object Model (POM). It states information about various aspects of the project, such as name, version and which dependencies the project has to other resourses, such as external libraries. An example of a POM can be seen in listing 4.2.

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>no.robert.lambdaprototype</groupId>
4    <artifactId>lambdaprototype</artifactId>
5    <version>1.0-SNAPSHOT</version>
6    <packaging>jar</packaging>
7    <name>lambdaprototype</name>
8     <dependencies>
9          <dependency>
10              <groupId>junit</groupId>
11              <artifactId>junit</artifactId>
12              <version>4.8.2</version>
13              <scope>test</scope>
14         </dependency>
15    </dependencies>
16 </project>
```

Listing 4.2: Project Object Model (POM)

Based on this POM, Maven will perform source compilation, download the dependent JUnit library, run automated unit tests and create a JAR-file.

By "dependent library", we mean that the project described by this POM depends on another project (library or framework). In this example, the project is dependent of the JUnit library. When a dependency is added

[3] http://maven.apache.org/

to a project, files from the project beeing added as dependency may be imported in classes in the project requiring the dependency.

An unlimited number of dependencies can be specified in the POM, and they will be automatically downloaded by Maven if they are not already present in the system. Note that Maven supports transitive dependencies. The developer does only need to specify those resources that the project directly depends on. If a directly dependent library has dependencies, these dependencies are handled by Maven automatically and transparently.

Maven downloads the necessary resources from a central Maven repository. However, the user can specify alternative download locations.

Fluenty has been created as a Maven project, and any developer wanting to develop it further should continue to use Maven as a the software management tool. Using Maven does to some extent force the developer to continue the development in a test-driven way. Maven can run all unit tests as a part of the building process, its build life cycle. If one test fails, the entire project will fail to build. Automated running of unit tests can be turned off, but to be consistent with the TDD principles that option should not be utilized.

4.3 Frameworks and techniques

This section presents the most central frameworks and techniques used in the development.

4.3.1 Invocation handling

Invocation handling is one of the most central and important concepts in Fluenty. In the example queries mentioned previously, we have seen that methods in the persistent entities is used directly in queries, like this:

$$...having(on(Book.class).getTitle()).eq(\text{"Booktitle"}));$$

For the user, `getTitle()` appears to be a regular method call. However, Fluenty does not use this method call to retrieve the title of a book. Instead, data derived about the invocation is utilized to determine which table column in the database that the constraint (equal) applies to. That is, in the column "title", which rows has a value equal to "Booktitle"?

Data about the method invocation is used also to ensure type safety. It is determined that since `getTitle` returns a String, `eq`'s parameter must be of a compatible type in order to make the query compileable.

Since we use `Book.class` as parameter to the method `on`, we use a `Class`-object[4] and not a `Book`-instance. But how can we then get access to methods in the class `Book`, and not only methods in `Class`? And how do we prevent an exception from beeing thrown, as `getTitle` is contained in `Book` and not in `Class`?

[4] http://download.oracle.com/javase/7/docs/api/java/lang/Class.html

The solution is to use a proxy object. The "Gang of Four"[12] describes that a "...proxy allows for object level access control by acting as a pass through entity or a placeholder object". A proxy can be seen as a gate. When a method invocation arrives, it is not allowed to continue to the real object, see figure 4.1. We can instead define an alternative behaviour, like registering data about the method invocation.

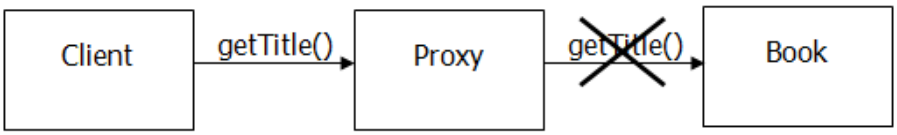

Figure 4.1: A proxy object

The creation of proxy objects is implemented by the method `on`. Naturally, this is a very central method in Fluenty. It returns a proxy object of the same type as its parameter. In the example query above, it returns a proxy object of class `Book`, which pretends to be a regular `Book`-object. Therefore, we can access all methods in class `Book`, while all method invocations to them is intercepted by the proxy instead.

A proxy object has no behaviour. Its behaviour is implemented by an invocation handler. Each proxy instance has an associated invocation handler object. The job of an invocation handler is to perform requested method invocations on behalf of the proxy.

This technique is very powerful, but underutilized [5]. The most typical usage is probably to change the behaviour of a class at runtime, e.g. to add a different implementation of `getTitle()`. In Fluenty, we don't want the method to be executed at all. That is, neither the original version nor a new implementation. We want the method call to be intercepted and instead capture data from the invocation. For that purpose, our invocation handlers use the Java Reflection API [6].

Reflection can observe and modify program execution at runtime, and can be used to obtain data about a method and its invocation by using methods contained in objects of type `Method`. Table 4.1 on the facing page presents a list of the captured data and its purpose.

With Reflection, Java has a native solution to create proxy objects and invocation handlers, in the packages `java.lang.reflect.Proxy` and `java.lang.reflect.InvocationHandler` respectively. However, this is limited to runtime implementation of interfaces. That is, if a class implements an interface, Reflection can provide a new implementation of that interface at runtime.

For Fluenty, this is not sufficient. Each persistent object is a JPA 2.0 entity. More information about this will be presented in subsection 4.3.6, but now it's only necessary to know that a JPA 2.0 entity is a

[5] http://java.dzone.com/articles/power-proxies-java
[6] http://download.oracle.com/javase/tutorial/reflect/

Data	Type
Name	The invoked method's name is used for mapping to a column in a database table. As an example, `getTitle()` will be mapped to column "title" (see chapter 5.1 on page 40
Return type	Based on the return type of the method, valid parameter types to the restriction method (e.g. `eq`) is determined.
Arguments	Currently not used. Included for possible future work.
Target type	The target type is the class object representing the class that declares the method. It is used to determine what object type the query eventually should return.

Table 4.1: Captured data about method invocations

POJO (previously discussed in section 2.1 on page 9). A POJO does not implement any interface. We therefore need to use a technique that allow us to work with proxies and invocation handlers without dealing with interfaces.

4.3.2 Code Generation Library

Code Generation Library (Cglib) is used to extend Java classes and to implement interfaces and dynamic subclasses at runtime[7]. It utilizes ASM's[8] mechanisms for bytecode manipulation.

Cglib does not require implementation of interfaces. Thus, it does not enforce the same limitations as Reflection. It is difficult to fully understand Cglib and the functionality it provides. Official tutorials does not exist, and documentation is limited to a few JavaDocs with very basic API documentation. Fortunately, its functionality is considerably better than its documentation.

To describe the usage of Cglib we will again use this example query:

$$...having(on(Book.class).getTitle()).eq("Booktitle"));$$

The method `on` returns a proxy object of the type that is provided as parameter. `on` is shown in listing 4.3.

[7] http://cglib.sourceforge.net/
[8] http://asm.ow2.org/

```
1    public static <T> T on(Class<T> type) {
2       if( METHODREF.get() == null ) {
3         MethodRef methodRef = new MethodRef();
4         METHODREF.set(methodRef);
5       }
6         return proxy(type, new InvocationRegistrar(METHODREF.get()
              ));
7     }
```

Listing 4.3: The method `on`

We see that that `on` is not responsible for the creation of the proxy object itself. That is delegated to the method `proxy`. A shortened version is presented in listing 4.4.

```
1  public static <T> T proxy(Class<T> type, Callback callback) {
2         Enhancer enhancer = new Enhancer();
3         enhancer.setSuperclass(type);
4         enhancer.setCallbackType(callback.getClass());
5
6     Class<T> proxyClass = enhancer.createClass();
7
8         T proxy = (T) ObjenesisHelper.newInstance(proxyClass);
9         ((Factory)proxy).setCallback(0, callback);
10        return proxy;
11 }
```

Listing 4.4: Shortened version of `proxy`

`Enhancer` is a Cglib class which is used to create dynamic subclasses. We define which class that should be subclassed on line 3. That is, which class to create a proxy of. `proxy` recevies a `Callback` as parameter. `Callback` is Cglib's expression for invocation handler. If we go back to listing 4.3, we see that this parameter contains an instance of `InvocationRegistrar`. `InvocationRegistrar` is Fluenty's invocation handler, and is an implementation of Cglib's interface `MethodInterceptor` (`MethodInterceptor` is a sub-interface to `Callback`). Cglib provides other interfaces, with different properties, that can be used as specifications for invocation handlers. `MethodInterceptor` is described as a "general-purpose callback", and was found appropriate for Fluenty.

As mentioned, each proxy has its own invocation handler instance. We define `InvocationRegistrar` as the proxy's invocation handler with the call to `setCallBackType` on line 4. On the succeeding lines, we instanciate the proxy object and return it back to `on` which in turn returns it to where `on` was invoked in the query. There it will seems like the proxy object is a regular `Book` object.

If we go back to the example query we see that directly after our invocation of `on`, we invoke `getTitle()`. This invoke will then be intercepted by `InvocationRegistrar`. It overrides one method from `MethodInterceptor`, `intercept`. This method intercepts any call to one of the proxy's methods.

The interception concludes Cglib's contribution. How the interception is treated will be described in the following subsections.

4.3.3 Thread-local variables

When an interception occurs, the data listed in table 4.1 is stored in a separate object. If we go back to listing 4.3, we see the instanciation of that object, `MethodRef`, occurs on line 3. After the instanciation, the object is passed as parameter to `METHODREF.set()`. `METHODREF` is a so-called thread-local variable. That variable holds a reference to a `MethodRef`-object, while each `MethodRef`-object in turn holds a reference to another `MethodRef`-object. That is, the thread-local variable points to a linked list of `MethodRef`-objects. One instance of `MethodRef` is created per method interception. Thus, each `MethodRef` instance contains data about one method invocation.

As the uppercase letters indicate, `METHODREF` is a constant. Thus, it is declared to be a static field. If we go back to listing 4.3 again we see that `on` is declared to be a static method. Hence, `METHODREF` must also be static, since a non-static field cannot be accessed in a statical way. However, this static approach raises some special issues that need to be addressed.

Most major applications, e.g. web applications, run in several threads. In Java, each thread has its own separate stack containing all local variables, parameters and return values. That is, they are thread-local. However, static fields are naturally not part of this stack, and is normally saved in a separate memory location on the heap. This location is shared among all instances of the same type. Thus, multiple threads can attempt to use the same variable (`METHODREF`) concurrently.

This problem has two possible solutions. One solution would have been to skip the static approach. Then, the object containing the method `on` must have been instantiated explicitly and used in the query. This was considered as an unnecessary step, generating unnecessary code violating the fluent syntax.

The other solution was to make the static variable thread-local. With Java, thread-local variables is created by using the class ThreadLocal[9]. It can be declared with the code given in listing 4.5.

```
1 private static final ThreadLocal<MethodRef> METHODREF = new
      ThreadLocal<MethodRef>();
```

Listing 4.5: Initilization of a thread-local variable

The variable's value can be set and retrieved with the methods `set()` and `get()` respectively.

With this technique we can keep the static references to `on` and `METHODREF`, and still keep `METHODREF` thread-safe. Thus, we can skip the unnecessary instanciation step and maintain the fluent syntax.

[9] http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ThreadLocal.html

4.3.4 Handling invocation chains

Each `MethodRef` instance corresponds to one method interception. In the example query below, the only method interception is for the invocation of `getTitle()`.

$$...having(on(Book.class).getTitle()).equal("Booktitle"));$$

In the query below we do however have a chain of invocations, with `getAuthor()` and `getName()` chained together.

$$...having(on(Book.class).getAuthor().getName()).equal("Author"));$$

As mentioned, Fluenty translates a query into a JPA 2.0 query - it uses JPA 2.0 to build queries internally. See subsections 4.3.5 and 4.3.6.

To be able to do this translation, we need to keep data about all method invocations that has occured in the same query. The query in the latter example above will result in a list of two `MethodRef` instances, each containing data about one invocation (`getAuthor()` and `getName()` respectively).

4.3.5 Query translation

Subsections 4.3.1 through 4.3.4 describe the preparation steps necessary to be able to execute a Fluenty query. "Execution" of a Fluenty query involves translation into a JPA 2.0 Criteria query and execution of that query. The actual execution is thus provided by the Java Persistence API. This technique was chosen among other alternatives, which will be discussed in subsection 4.3.6.

The core building block in the query translation is the captured data about method invocations, listed in table 4.1. Listing 4.6 shows a simplified example of how this data is utilized in the translation. On line 1, we see that the invoked method's target type is used to determine the type of the query root. A query root is similar to the FROM clause in a SQL query [26]. On the next line, the method's name is used to create the query path. A path is the result of navigation from the root expression to one of its attributes [26]. The method's name is used to determine the corresponding table field for the value, e.g. will a method named `getTitle()` be mapped to the field `title`. This will be discussed further subsection 5.1.1 on page 39.

The return type is not directly used in the query translation. However, it is important when handling invocation chains. If the return type is `void`, then there is no more methods in the chain.

```
1 Root<T> root = criteria.from(methodRef.getTargetType());
2 Path<Object> path = root.get(asProperty(methodRef).getName());
3 criteria.select(root);
4 criteria.where(builder.equal(root.get(asProperty(methodRef).
    getName()), propertyValue));
```

Listing 4.6: Simplified example of query translation

This example is for illustration purposes only. The query translation in Fluenty is more complex. It supports other constraint methods than just for determination of equality, as well as logic for handling collection properties and longer invocation chains, among others. Some steps both before and after the example have been omitted.

4.3.6 JPA in Fluenty

Translation of queries into a JPA 2.0 Criteria query was not the only possible option. The same result could possibly been achieved using another API like Hibernate Criteria, and also SQL.

It was discovered early that choosing SQL would have implied unneccessary work. An API facilitates well for using variables (e.g. variables from `MethodRef`) directly in queries, and it also has some type safety mechanisms built in. This is features that we could take advantage of, instead of having to write an own implementation as to achieve such functionality, as if we had chosen SQL.

It was not desireable to rely Fluenty's functionality on any third party vendor. This excluded APIs like Hibernate Criteria.

There are several persistence techniques available for Java. JDO and JPA, are "standardized" through the Java Community Process Program (JCP)[10]. JDO supports a larger selection of datatypes (from different Java class libraries) and does not have the same restrictions on classes describing persistent objects. As listed in table 4.3 on the following page, JPA 2.0 supports neither final classes annotated as entities nor final methods in those classes.

JDO does not have these restrictions. Generally it provides a more comprehensive specification than JPA, and it would have implied fewer limitations to Fluenty than JPA 2.0 does. However, a major and decisive drawback is that JDO does not have an accompanying query API. It is available as an extension to QueryDSL (section 2.2 on page 12), but again, to rely Fluenty on a third party vendor was not desireable. JPA 2.0 therefore remained as the only actual choice.

That Fluenty "produces" JPA 2.0 Criteria query objects facilitates both easy further development, as well as the ability to integrate Fluenty seamlessly into projects already using JPA. Many of the current ORM-tools support JPA. Therefore it will be easy to integrate Fluenty with an existing tool (research question 3 in section 1.6 on page 6), or use them interchangeably within the same application (figure 4.2).

JPA 2.0 raises some requirements that persistent objects in applications using Fluenty need to fullfill. They are defined in [25], and presented in table 4.2, while table 4.3 contains a listing of the most central features and constraints introduced by JPA (a subset from the JPA specification [11]).

[10] http://www.jcp.org

[11] http://db.apache.org/jdo/jdo_v_jpa.html

Requirement	Description
Annotations	The class must be annotated with the `javax.persistence.Entity` annotation.
Constructors	The class must have a public or protected, no-argument constructor. The class may have other constructors.
Final declaration	The class must not be declared final. No methods or persistent instance variables must be declared final.
Serialization	If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
Inheritance	Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
Access types	Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class' methods. Clients must access the entity's state through accessor or business methods.

Table 4.2: JPA 2.0 entity requirements

Feature	Support
JDK Requirement	1.5+
Persistence specification mechanism	XML, Annotations (preferred from JPA 2.0)
Datastore	RDBMS only
Restrictions on persisted classes	No final classes. No final methods. Non-private no-arg constructor. Identity Field. Version Field.
Persist static/final fields	No
Transactions	Optimistic locking
Query language	JPQL, SQL
Criteria API	Yes
RDBMS Schema Control	Tables, columns, PK columns, FK columns, unique key columns
ORM Relationships	1-1, 1-N, M-N, Collection, Map

Table 4.3: JPA features

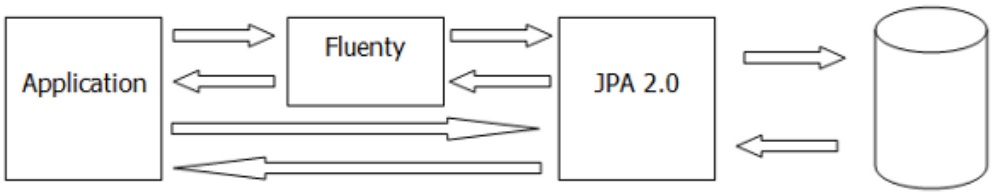

Figure 4.2: Fluenty and JPA

4.4 Implementation challenges

This section describes the parts of the implementation which have been most challenging.

4.4.1 Collection properties and type safety

The techniques just described proved to not be sufficient with certain queries. A problem query is presented in listing 4.7.

```
1 List<Book> books = repository.find( Book.class, having(on(Book.
      class).getAuthors()).having(on(Author.class).getName()).equal(
      "Rune Flobakk" ));
```

Listing 4.7: Typesafety along call chain

This query returns all books that have an author with a certain name. The problems apply if a book has several authors. Technically, this will typically mean that each `Book`-object has a `Collection` of `Author`-objects.

The encountered problems can be summarized as follows:

- **Return type**. The last part of the query can be considered as a subquery. It returns `Authors`. However, since the `List` is parameterized with `Book`, that is the expected type. Due to the evaluation order it turned out to be more complicated than originally assumed to return objects of the correct type.
- **Parameter types**. Type safety between parameters must be attained several places within the same query, as well as between different data types (`Books` and `Authors`)
    1. The get-method after the first `on` (`getAuthors()`) must return objects of the same type as the parameter to the last `on`. It needs to be verfied that the objects we want the subquery to return (`Authors`) actually are contained in the container object (`Book`).
    2. The type of `eq`'s parameter must be applicable to the type returned by the last `on`'s get-method (`getName()`).

In an attempt to solve these problems, several solutions were developed. However, it turned out to be more challenging than expected to create one that solved each of the mentioned problems at the same time.

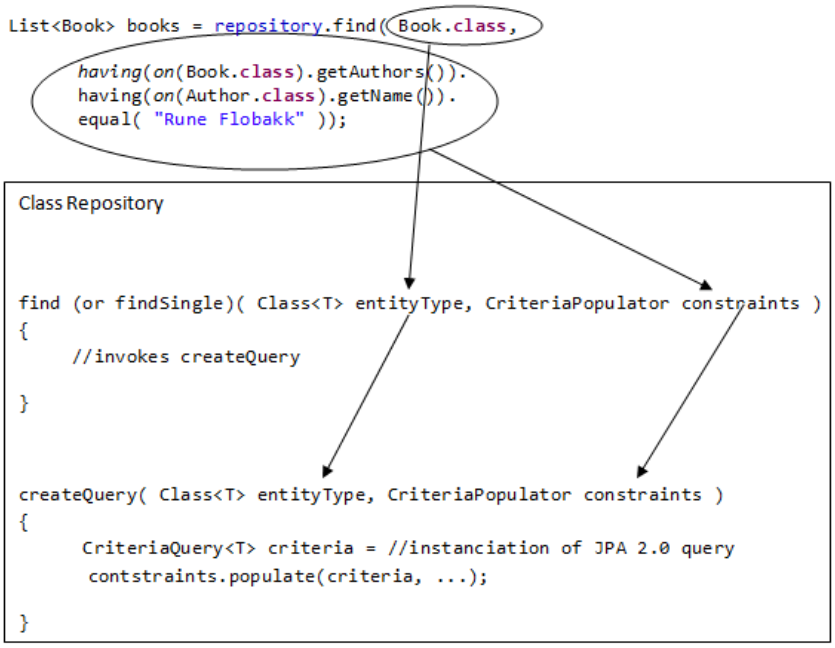

Figure 4.3: Mapping from query to parameters and methods in Fluenty

We see in listing 4.7 that `find` has two parameters, where the last is the value returned from `having`. Further, we see that we have two invocations of `having`. The first takes the invocation to `getAuthors()` as parameter, while the last takes `getName()`. As described in subsection 4.3.1, the actual invocation is never performed, instead data about the invocation is captured and stored.

We have chosen to make two implementations of `having`. One is to be executed if its parameter returns a collection (like `getAuthors`), while the other will execute if the parameter is of a single type (like a single property in the object). The two types of parameters require different behaviour. If a collection is received, an object of type `CollectionPropertySpecifier` is returned. The opposite is `SinglePropertySpecifier`. When initialized, both need a reference to the MethodRef-chain, described in subsections 4.3.3 and 4.3.4, and a reference to the previous specifier.

With "previous specifier", we mean an object of one of the two object types just mentioned. First time this happens there will naturally not exist any previous specifiers. However, if we go back to listing 4.7, we see that `having` is invoked twice. The first `having` will return a `CollectionPropertySpecifier`. Since `getName()` naturally returns a single object, the last `having` will return a `SinglePropertySpecifier`. This will then have a reference to the previous specifier; we will thus have a chain of two specifiers.

Both specifiers share a common interface, `CriteriaPopulator`, containing a method `populate`. This method performs the construction of the JPA 2.0 Criteria query (see subsection 4.3.6).

Figure 4.3 shows how elements from the query map to the different parameters and methods mentioned above. The last `having` returns as `SinglePropertySpecifier`, and this specifier contains the restriction method `equal`. `equal` returns the `SinglePropertySpecifier` instance, which is the parameter `constraints`, shown in the figure. We see further in the same figure that the instantiated JPA 2.0 Criteria query is passed as parameter to `populate`, where the first thing that happens is an invocation to the previous specifier's `populate`. In this case, that will be a `CollectionPropertySpecifier`.

Both specifiers will then modify the same JPA 2.0 Criteria query instance. The `CollectionPropertySpecifier` will specify the query root (see subsection 4.3.5) to be `Book`, on the basis of the target type from `MethodRef` (see subsections 4.3.1 and 4.3.3). The root decides the type of the whole query. Here we benefit from a mechanism in JPA 2.0 Criteria, which allows roots to be parameterized with generic types, i.e. `Root<T>`. `populate` takes an instance of `CriteriaQuery<T>`. We see in figure 4.3 that the class object stating the entity type is also parameterized with `<T>`. Since the entity type is `Book`, the `CriteriaQuery` will be on books. Then, trying to assign the return values from the query to a list parameterized with something else than `Book` will result in a compile error. Thus, we have addressed the first bullet point in the previously mentioned list of encountered problems.

Further, the `CollectionPropertySpecifier` sets the path to be to authors on the basis of information stored in the `MethodRef` chain. Again, see subsections 4.3.1 and 4.3.3. See subsection 4.3.5 for a more thorough description of a path.

Then, `populate` in `SinglePropertySpecifier` will create a `Subquery` of the same `CriteriaQuery` instance. Its root is determined from the `MethodRef` chain. This subquery will select `Authors` with the given name. Finally, the `CriteriaQuery<T>`-instance selects those books that have a collection (the path) containing those elements returned from the subquery.

By doing this separation in specifiers we have been able to also solve the problems described in the last bullet point in the list of problems. It becomes easy to separate the query into essentially two queries, CriteriaQuery and Subquery. Thus, the mechanisms used to ensure type safety in simpler queries can be used for such more complex queries as well. Additionally, since the specifiers are chained together, and thus maintains a correlation between the queries, type safety between queries can also be achieved. That is, to e.g. ensure that each `Book` has a collection of the type specified in the last part of the query.

4.4.2 Single properties in chain

Refer to this example query:

$$having(on(Book.class).getPublisher().getName()).eq("Addison-Wesley"));$$

Here, `getPublisher()` is a method in the class `Book`. So, when the method `on` is invoked, an invocation handler capturing invocations to

methods in class `Book` will be created. Then, `on` simply returns `null`. However, if the get-metod returns a primitive type, returning `null` will create a NullPointerException, because primitive types cannot have a `null`-value. Therefore, the return type of the method must be determined. If it is a primitive type, a default value of the determined type will be returned instead of `null`.

In the example query above, the invocation to the method `getName()` in `Publisher` is intercepted. So, here we have two expected types, `Book` and `Publisher`. Since a proxy can be used to intercept method invocations to methods in one class only, two proxies are needed. But, if the approach described above, with returning either `null` or a primitive type, is used here, it will be impossible to create more than that first proxy object for `Book`.

Therefore, it does not suffice to check whether the return type is a primitive type or not. If the return type is neither primitive, nor a final class, then a new proxy must be returned. This proxy must be of the same type as the return type, so it can record invocations to methods in that class.

When the invocation handler is created for `getPublisher()` it will examine the return type and conclude that its return type is neither primitive nor final. It will then return a new invocation handler of type `Publisher`, which captures the call to `getName()`. When the return type of that get-method is examined, it will be recognized as of type String, so no new invocation handler is needed.

Chapter 5

Design

This chapter focuses on the usage of design patterns and why they have proven beneficial when developing Fluenty. A discussion of other design choices, both from a technical and a user perspective, is provided, as well as an overview of central classes and their associations.

5.1 Design patterns

As an attempt to make the development process efficient and Fluenty robust and reliable with few errors, we have been inspired by appropriate design patterns. This improves the readability of the code, and subtle issues that can cause major problems can be avoided [12]. This is not just relevant for creating object-oriented software. Christopher Alexander described patterns in buildings and towns. His definition, used by "The Gang of Four" (GoF) [12], stated that *"each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*.

The succeeding subsections will list the design patterns that have been inspiring when developing Fluenty. Additionally, a description of a possible new pattern that has been identified during the work will be presented.

5.1.1 The Strategy pattern

As mentioned repeatedly, the user can use methods in the persistent objects directly when writing queries. Typically, this will be get-methods as in the example query in listing 5.1.

```
1 List<Book> books = repository.find(Book.class, having(on(Book.
      class).getTitle()).equal("How To Be Awesome"));
```

Listing 5.1: Simple example query

The get-method returns a property, a field, of a persistent object, in this case the title of a `Book`. That get-method is naturally not present in

a database table. With ORM, a column name in the database table will typically be mapped to a field name in the persistent object. In Fluenty, we therefore need to be able to resolve a field name on the basis of a method name.

Currently, the JavaBean[1] convention lays the foundation for resolving field names in Fluenty. The resolving strategies currently supported is listed in table 5.1.

Method name	**Field type and name**
`getSomething()`	`<type> something;`
`isSomething()`	`boolean isSomething;` or `boolean something;`
`hasSomething()`	`boolean isSomething;` or `boolean something;`

Table 5.1: Field resolving strategies

`<type>` will depend on the method's return type. Even if this strategy currently fits Fluenty quite well, other users might have other demands regarding how the resolving should be done. To make a design that facilitates this, the implementation of the mapping algorithm was done according to a design pattern called "Strategy". This pattern has been described by GoF [12] as suitable if different variants of an algorithm might be needed.

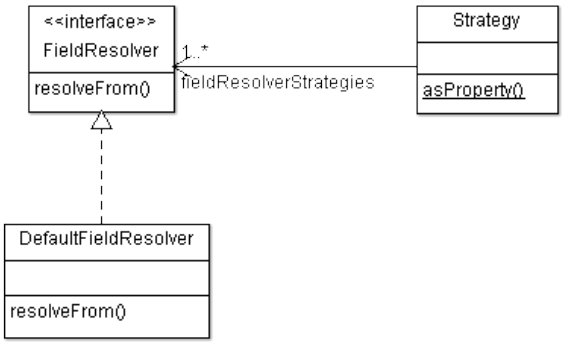

Figure 5.1: Strategy pattern in Fluenty

The class `Strategy` in figure 5.1 contains a static method `asProperty`. This method is invoked when translating a query into a JPA 2.0 Criteria query in the earlier mentioned specifier-classes (SingleProperty and CollectionProperty). It takes a `MethodRef`-object as parameter. The properties of a `MethodRef`-object have been discussed earlier, in subsection 4.3.3 on page 31. It contains both the target type and the invoked method's name. The target type is used to determine whether the resolved field is valid or not. This can be illustrated with an example. Let's say that the target type is `Book`, and the method's name is `getTitle()`. That is, the user has written `on(Book.class).getTitle()` in the query. Then, the current resolving

[1] http://en.wikipedia.org/wiki/JavaBean

strategy will look for a field named `title` in class `Book`. If found, this field is returned. Otherwise, an exception will be thrown since the query attemts to ask for a non-existing property (field).

Note that the relationship between `Strategy` and `FieldResolver` is one-to-many. Several implementations of resolveFrom(), that is, several strategies (algorithms), can be attempted in order to resolve the field name. If an additional strategy is needed, a new implementation of the interface `FieldResolver` must be added.

The isolation of the resolving strategy in a separate object has proven convenient when doing TDD. We have created separate test suites, testing only the field resolving functionality without any concerns about the rest of Fluenty's functionality.

5.1.2 The Facade pattern

Larman[21] describes a facade as a "front-end" object. It's a single point of entry to the services of a subsystem. The subsystem is hidden behind that object. The term subsystem just indicates a separate grouping of related components. Thus, it is used in an informal sense, and not exactly as defined in the UML standard.

We believe that the starting point for every Fluenty query, an object of type `Repository`, can be considered as Fluenty's facade. It contains two methods, `find` and `findSingle`. Both methods execute the query and return the query result. As we have seen in previous code listings, they take an entire Fluenty query as parameter. Actually, this is a JPA 2.0 Criteria query object (see subsections 4.3.5 and 4.3.6). However, the user does not need to relate to the `CriteriaQuery` and the process of translating the Fluenty query to a JPA 2.0 Criteria, as this is hidden behind the facade.

5.1.3 The Singleton pattern

When using the singleton pattern, the application should use exactly one instance of a class. In Fluenty, the object of type `Repository` is an example of a class that requires only one instance. However, this is not strictly necessary, as Fluenty will work correctly regardless of how many instances have been created. Therefore, we have not implemented any logic to ensure that `Repository` is instantiated only once, i.e. as a singleton.

However, the object `MethodRef` is an object that must be a singleton. As discussed in subsection 4.3.3 on page 31, the `MethodRef`-objects form a linked list. The method `on` in `MethodRef` does therefore add a new `MethodRef` object to the thread local variable if, and only if, no object has been added to the thread local variable previously. Naturally, if a new object is added to that variable each time `on` is invoked, the linked list will disappear.

We see the principle in listing 5.2. We check whether the thread local variable has yet been assigned a value. If so, that value is retrieved and used as parameter when instantiating `InvocationRegistrar`. Otherwise,

a new `MethodRef`-object is assigned to the variable prior to the instantiaton.

```
1    public static <T> T on(Class<T> type) {
2       if( METHODREF.get() == null ) {
3         MethodRef methodRef = new MethodRef();
4         METHODREF.set(methodRef);
5       }
6         return proxy(type, new InvocationRegistrar(METHODREF.get()
              ));
7     }
```

Listing 5.2: Singleton

5.1.4 The Proxy pattern

The usage of dynamic proxies, described in section 4.3.1 on page 27, follows some of the principles of the Proxy pattern. This pattern was first described by the GoF [12]. Fluenty does not utilize the pattern as they described it. Instead, a summary of their description of the pattern is provided here, as background information for the next subsection.

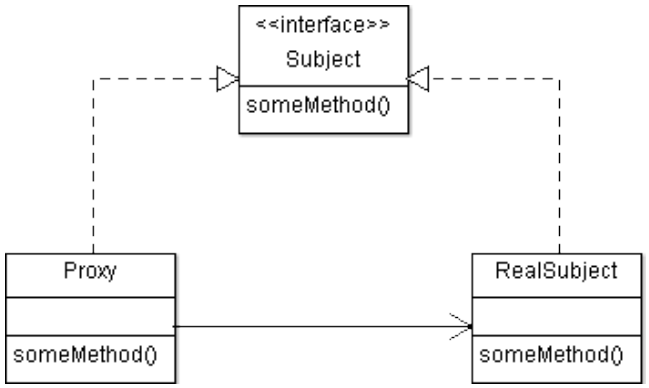

Figure 5.2: Proxy pattern

The pattern is one of the simplest described by the GoF [12], yet it's one of the most underutilized [13]. It seems like many developers is unaware of its power. According to GoF, it *"let you change the real behaviour from a caller point of view since method calls can be intercepted by the proxy"*.

What a proxy is and its purpose has been described earlier in this report, in subsections 4.3.1 and 4.3.2. Figure 5.2 shows how GoF [12] have described the structure of the proxy pattern. We see that they define it to have three participants; the proxy, which is a substitute for the object "RealSubject", and "Subject". The object's name is not important here, "RealSubject" could be an object of any type. More important, the proxy and the real subject share a common interface, "Subject". Thus, the proxy object can be a substitute for the "RealSubject"-object.

5.1.5 New pattern suggestion – invocation handling pattern

As just discussed, Fluenty's usage of dynamic proxies follows some of the principles from GoFs description. But, it differs in at least two aspects:

1. A proxy object created in Fluenty does not share a common interface with the proxied object.

2. The proxy does not forward requests to the proxied object, neither does it control access to the object since the proxied object is not used at all. In GoF's description, the proxy object acts as a substitute for the real object just until the real object is actually needed.

These differences, in addition to our combination of proxy objects with thread-local variables to create a fluent API, reveals the identification of a possible new pattern. We are aware of only one tool that utilize some of these techniques in production code. That is LambdaJ[2], which we have mentioned earlier in this report.

Subsections 4.3.1, 4.3.2 and 4.3.3 describe how this functionality has been developed, and only a brief summary is provided here. The most important classes is `InvocationRegistrar` and `MethodRef`, as well as the interface `InvocationRegistry`. `InvocationRegistrar` is a subclass of `MethodInterceptor`, which is part of Cglib, described in section 4.3.1. By extending that class, `InvocationRegistrar` becomes a method interceptor which registers data about the actual invocation that triggered the interception. The handling of the invocation data is delegated to the `InvocationRegistry`, an interface implemented by `MethodRef`. These classes are included in the class diagram showing the most important classes and their relationships in figure 5.3 on page 47.

All functionality regarding registration of method calls and data about them has been put in a separate package. By isolating that functionality, it can be re-used in other projects with a different purpose than Fluenty. A discussion about other areas where proxy objects can be a useful contribution (research question 4) can be found in section 6.4 on page 61.

In the design we attempted to follow the Single Responsibility Principle[3]. This is a principle of class design stating that a class should have one, and only one, reason to change. As the term implies, each class should have a single responsibility, and this responsibility should be completely encapsulated by that class.

This makes the behaviour of `InvocationRegistrar` concise and unambigous. It also makes it simple to test the behaviour in isolation by "mocking" in an `InvocationRegistry` and observe (during TDD) if it behaves as expected.

As a digression, "mocking" and "mock objects" are central terms when working with unit testing and TDD. A mock object is a simulated object. In controlled ways, it imitates the behaviour of a real object. It can be

[2] http://code.google.com/p/lambdaj/

[3] http://en.wikipedia.org/wiki/Single_responsibility_principle

compared to a crash test dummy, that simulates the behaviour of a real human beeing when crash testing vehicles [4]. Sometimes, it is not practical to use real objects in a unit test. This might be due to performance reasons, the object may not exist at the time of test, it may change behaviour, or it may contain information only relevant for testing purposes. Then, a mock object is "mocked" into the tested code instead. The framework Mockito[5] was used to create such objects when developing Fluenty.

5.2 What is good API-design?

It is claimed in [15], that to create a bad API is easy, while to create a good is difficult. Further, some of the consequences and effects of poor APIs are discussed, where some are listed in table 5.2.

Issue	**Consequence**
Hard to understand	Poor APIs are difficult to understand and use. Thus, writing code against poor APIs requires more time. This lead to increased development costs.
Additional code	Makes programs unnecessary large and less efficient.
Complex code	The additional code may lead to unnecessary complex code more prone to bugs. This increases testing effort, hence the costs, as well as the risk for bugs to remain undetected even after the application has been deployed.

Table 5.2: Effects and consequences of poor APIs

The usage of design patterns, discussed in section 5.1, focuses on how to create a good design from a technical point of view. It can help in creating a robust and reliable API, which of course is appreciated by the users. However, beeing robust and reliable does not affect the issues listed in table 5.2. To try to address these issues as well we have followed some relevant guidelines for API design. These guidelines are described in [15]. It is claimed that following these guidelines is not a guarantee of success, but taking them into account makes it more likely that the result will turn out to be usable.

An API should be minimal, without imposing undue inconvenience to the caller

"The fewer types, functions and parameters an API uses, the easier it is to learn, remember and use correctly" [15]. We have tried to keep the number of elements that the user needs to relate to at a minimum. When designing, we have separated Fluenty's functionality into three parts, (1) preparations before a query can be written, (2) query writing, and (3) usage

[4]Example from Wikipedia: http://en.wikipedia.org/wiki/Mock_object
[5]http://code.google.com/p/mockito/

of the query results further in the code. By doing this separation we believe it has been easier to identify those elements that are absolutely necessary for each part, than if we had looked at Fluenty as a whole. We list the key points why we believe Fluenty adheres to the ideal of minimalism below.

1. Fluenty introduces a new type, `Repository`. This needs to be instantiated, and it's `EntityManager` must be set before a query can be written. The `EntityManager` is not a new type for the user, as he has already been introduced to it by using JPA 2.0 (see subsection 6.1.2 on page 51 for a more thorough description of an `EntityManager` and its purpose). As described in 5.1.2, the `Repository`-object is the single point of entry to Fluenty, and is the only new type introduced to the user by Fluenty that needs to be explicitly instantiated.

2. When writing a query, the user is introduced to three new methods, `find`, `having` and `on`. They are present in every query. Thus, the user should be able to familiarize with them rather quickly. In addition, a query consists of one or more constraint methods, like `equal`, `greaterThan`, etc. Their purpose is considered as self-explanatory.

3. Queries return persistent objects without any additional method invocations or type conversion beeing necessary. Thus, no new types or functions are introduced to the user in this part.

APIs should be documented before they are implemented

Writing documentation afterwards, by the developer, tends to result in just a description of what he or she has done [15]. This can lead to incomplete documentation and an API that does not fullfill its requirements. The developer has had a large focus on implementation issues, and requirements and desires of the customer/initiator might come second. To avoid that from happening in Fluenty, requirement identification and specfication have been a cooperative task between project initiator and developer. It resulted in the practical usage description in section 3.3 on page 19 and the user stories in appendix A. This served as documentation prior to the implementation.

We have strived to make Fluenty as intuitive as possible. Thus, we tried to eliminate the need for documentation as much as possible. Important aspects in that regard is fluent interfaces, queries construction using methods in the persistent objects and keywords close to natural language.

Good APIs don't pass the buck

As described in [15], a way to "pass the buck" when designing an API is to be afraid of setting a policy. "Well, the caller might want to do this or that, and I can't be sure which, so I'll make it configurable". This often results in a large number of methods, or complex methods taking five or ten parameters, making them cumbersome to understand and use.

We have strived to be clear about what Fluenty should and should not do, trying to avoid to end up with more complexity than necessary. The main instrument we have used is to limit the number of methods available

to the user at a given time, with the method chaining technique. This technique is described in subsection 3.2.3 as an important technique when talking about fluent interfaces. If the method `having` returns an object of type `SinglePropertySpecifier`, which in turn contains a method `equal` we can naturally write `having().equal()`. By limiting the number of available methods in each object to a minimum, we leave the user with few choices about what he can do next.

5.3 Structure

Figure 5.3 on the facing page presents an overview of central classes and their relationships. Classes with little impact on the core functionality, like exception classes, is omitted from the diagram. The same are packages containing JUnit test suites.

See appendix B for more information about how to gain access to the source code.

Figure 5.3: Central classes and their relationships

Chapter 6

Answer to research questions

In this chapter we will try to answer the research questions presented in section 1.6 on page 6.

6.1 RQ 1. What are the main differences, in terms of expressiveness, between Fluenty and other ORM query techniques?

In this context, expressiveness has been used as an umbrella term meaning readability and understandability, amount of work needed to write a query, as well as functionality. To be able to measure it we need to define the term more precisely.

Some studies evaluating language expressiveness and functionality have chosen a quantative approach. In this report, a qualitative approach has been chosen. Since Fluenty at the moment only is a POC implementation, its functionality is currently limited compared to existing techniques and tools. However, the principles and ideas behind it facilitates for expanding this in the future. Therefore, to gain more focus on principles, rather than current functionality, a qualitative approach was chosen. Queries written with Fluenty will be compared to queries written with Hibernate Criteria, JPA 2.0 Criteria and QueryDSL. These tools will be referred to as techniques.

In addition to the tools just mentioned, a fourth tool, Squeryl, was presented in the listing of existing solutions in section 2 on page 9. Squeryl is written for the Scala language. Thus, it has a syntax quite different from the other mentioned tools, and has therefore been omitted from the evaluation. Both Hibernate and JPA has its own string based query language. Since the differences between using a string based language and a query API have been covered earlier in this report, the string based techniques have been omitted from the evaluation as well.

The evaluation of the techniques will concentrate on the areas listed in table 6.1.

Criterion	Description
Preparations	What kind of installation and configuration needs to be done before queries can be written and executed?
Query writing	How is a query written? Using the programming language, an API, or a String-based language? Tool support?
Functionality	The evaluation is based on developers having complete functionality when writing a query with SQL, that is, full SQL-functionality. Elements like recursion is not supported by SQL. Queries written with any technique is at some point transformed into a SQL query. Something that cannot be expressed in SQL cannot be expressed with any other technique either. Hence, any other technique has functionality equal to or less than SQL.
Understandability	How closely the technique's keywords and constructs are related to English language. We assume absolute understandability for natural English language. Thus, the clearer relationship between English and the technique, the higher degree of understandability.
Result handling	How can the retrieved result be used further in the program code?

Table 6.1: Evaluation criterias

6.1.1 Scenario

The evaluation is based on how the techniques solve an identic task. The task is to retreieve all `Books` having an `Author` with a certain name. Prior to the evaluation, a domain model consisting of four domain objects was created. The model is shown in figure 6.1.

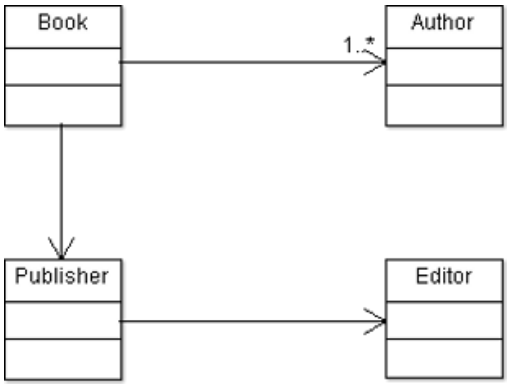

Figure 6.1: Domain model

HQSLDB was chosen as database technology. It's a simple, lightweight RDBMS written in Java. It runs in-memory and needs no extra infrastructure and little configuration to run.

Storage of objects was not part of the evaluation. However, in situations

where the way the objects are stored affects the preparations, this will be discussed as well.

Queries fall into four different categories: Select, aggregate, update and delete. This evaluation focuses only on the select-category, since this is the only supported by Fluenty, even if some of the techniques supports queries in several categories.

The terms ”project” and ”dependency” is used below several times. In this section, we refer to project as a development project, typically in the IDE (integrated development environment). By dependency, we mean that a project uses another project, library, etc. This was described in connection with Maven in section 4.2.1 on page 26. When a dependency is added to a project, files from the project beeing added as dependency may be imported in classes in the project requiring the dependency.

6.1.2 Preparations

By preparations, we mean everything that must be done before we can start to write and execute a query. Examples are creating configuration files, annotations and package imports.

Fluenty

Each class desribing a persistent object must be annotated with JPA 2.0 Annotations (see subsection 2.4.1 on page 14). Lack of annotations in any of the persistent objects used in a query will cause the whole query to fail. Unfortunately, this will not be discovered until runtime.

JPA 2.0 uses a configuration file, which must be named `persistence.xml` and placed in a folder named `META-INF` in the project root folder. In this file, a unique name for each persistence unit must be defined. A persistence unit defines the set of all classes that are related or grouped by the application. Additionally, database connection parameters must be specified by this file.

The final step is to create an object instance of `Repository`, mentioned in subsection 5.1.2 on page 41. This instanciation requires a parameter of type `EntityManager`. `EntityManager` is a JPA class associated with a persistence context. A persistence context is a set of persistent entity instances. Within the persistence context, the entity instances and their lifecycle are managed. The `EntityManager` instance is retrieved based on the persistence unit name defined in `persistence.xml`.

Fluenty must be added as a dependency in the project where it is used. The easiest way is to specify it in the Maven POM, since Fluenty is tailor-made to be used as a Maven project. Fluenty requires import of the package `no.robert.repository` and `no.robert.methodref` in each class where queries are written. If not, it will be impossible to use the methods `find`, `on` and `having`, which are key building blocks in any query. Additionally, the JPA classes
`javax.persistence.Persistence` and `javax.persistence.EntityManager` need to be imported as well.

Hibernate Criteria

The preparation steps are very similar to the ones described for Fluenty. However, there is an alternative to JPA annotations. The same metainformation can be defined by XML-files. One file must be created for each persistent domain object type, named `<DomainTypeName>.hbm.xml`. If this approach is chosen, database connection parameters must be specified in a separate configuration file named `hibernate.cfg.xml`. All configuration files must be placed in a folder named `META-INF`.

As with Fluenty, it is easiest to let Maven add a dependency to Hibernate. Access to Criteria methods is gained through the import of `org.hibernate.criterion.*`.

JPA 2.0 Criteria

Since Fluenty must be used on top of JPA 2.0, the preparations steps are almost identical. The only difference is the import of the package `javax.persistence.criteria.*` instead of the two `no.robert`-packages for Fluenty.

QueryDSL

The preparation steps are quite similar as for the previously mentioned techniques. A dependency needs to be added, preferably with Maven, and required imports must be made.

As mentioned in 2.2 on page 12, QueryDSL can be used with different backends. The backend in use determines which project should be specified in the POM and which imports are necessary. The package `com.mysema.query` (with sub-packages) must be imported in every class using QueryDSL. Additionally, there are packages containing elements specific for each backend technology, e.g. `com.mysema.query.jpa`, `com.mysema.query.jpa.hibernate` and `com.mysqma.query.sql`.

Since QueryDSL supports various backends, all persistence specification mechanisms mentioned previously, like JPA 2.0 annotations or Hibernate mapping configuration, can be used.

6.1.3 Query writing

This section will present how queries, to accomplish the task described in subsection 6.1.1, can be written with the four different techniques.

All code presented within listings is compileable, assuming that necessary dependencies are added to projects where they are used. All code has been tested in order to verify that it returns the desired results. Each query should be considered as a solution proposal only. Probably, the same result can be achieved by constructing the queries differently, using different query API methods or keywords.

Fluenty

```
1 List<Book> books = repository.find( Book.class, having(on(Book.
    class).getAuthors()).having(on(Author.class).getName()).equal(
    "Rune Flobakk"));
```

Listing 6.1: Query with Fluenty

Listing 6.1 shows the query only. The query assumes that the `Repository`-object is already instantiated. It starts by invoking `repository`'s method `find`, with the entire query as parameter. The first parameter to `having` indicates what type of objects we want the query to return. Then, we indicate which property in `Book` we want to retrieve the books on the basis of, namely its authors. After closing the first `having`'s parenthesis, we can start a "new" query with another invocation to `having`. This is constructed similarly to the first query. Except at the end, where we invoke the constraint method `equal` instead of another invocation to `having`.

We will receive a compiler error if `find`'s first parameter is not compatible with the parameterized `List`. Similarly, we will receive errors if `getAuthors` and `getName` is not contained in classes `Book` or `Author` respectively. The parameter type of the constraint method `equal` must be compatible with the return type of `getName`. Additionally, the compiler complains if `getAuthors` returns another type than specified by the parameter to the last `on`.

If we use a modern IDE, we have a high degree of tool support when using Fluenty. The compiler will give us immediate feedback if we write something wrong concerning the issues just described. Additionally, since we use the persistent objects' classes and their methods directly without any strings involved, we can take benefit of the IDE's functionality for refactoring and auto-completion of program code.

Hibernate Criteria

General principles and concepts regarding query writing with Hibernate Criteria was described in subsection 2.1.2 on page 11. Therefore, only a short description will be given below, with a more technical focus specific for this particular task.

```
1 Session session = (Session) entityManager.getDelegate();
2 Criteria criteria = session.createCriteria(Book.class, "book");
3 criteria.createAlias("book.authors", "authors");
4 criteria.add(Restrictions.eq("authors.name", "Rune Flobakk"));
5 List<Book> books = criteria.list();
```

Listing 6.2: Query with Hibernate Criteria

We start the query by retrieving the current session. To show that also Hibernate Criteria, not just JPA 2.0 Criteria, can be used together and interchangeable with Fluenty, we retrieve the current session with the JPA 2.0 Criteria `EntityManager`'s method `getDelegate`. This is the same

EntityManager we discussed when we described the preparation steps for Fluenty in subsection 6.1.2. Further, we instantiate the criteria query and create a so-called alias, to indicate which property in Book we want to retrieve the books on the basis of, before we add the eq-method as Restriction.

In a modern IDE we can benefit of auto-completion of method names in the Hibernate Criteria API. However, we specify other properties, like book.authors and authors.name, as strings. Thus, the IDE is unable to auto-complete them, and the compiler cannot detect any errors related to them. Examples are to specify a non-existing property, or to try to compare authors.name to another type than String. Additionally, if we refactor a persistent object's class, we must manually check all our queries and change the property name we have written in the string. An IDE will be unable to detect, and perform, the change if we just use its automatic refactoring functionality.

JPA 2.0 Criteria

```
1  CriteriaQuery<Book> criteria = criteriaBuilder.createQuery(Book.
       class);
2  Root<Book> root = criteria.from(Book.class);
3  criteria.select(root);
4  Path authors = root.get(Book_.authors);
5  Subquery<Author> subquery = criteria.subquery(Author.class);
6  Root<Author> subroot = subquery.from(Author.class);
7  Path name = subroot.get(Author_.name);
8  subquery.select(subroot).where(criteriaBuilder.equal(name,"Rune
       Flobakk"));
9  criteria.where( criteriaBuilder.isMember( subquery , authors));
10 List<Book> books = entityManager.createQuery(criteria).
       getResultList();
```

Listing 6.3: Query with JPA Criteria API

Listing 6.3 shows the query only, and assumes that criteriaBuilder and an EntityManager is already instantiated properly. See subsection 6.1.2 for a description of an EntityManager. Query roots and paths have been discussed in subsection 4.3.5 on page 32. We see that we use a Subquery to retrieve Authors with the specified name, before we narrow the outer CriteriaQuery with the results from the Subquery on line 9.

The structure we selected for the JPA 2.0 Criteria query differs from the one we selected for the Hibernate Criteria query in listing 6.2. In the Hibernate Criteria query we use a join while we in the JPA query use a subquery. In Fluenty, parts of certain queries is translated into JPA 2.0 Criteria subqueries, when needed, instead of joins. See subsection 4.3.5 for a discussion about the query translation process. Subqueries proved to be easier to integrate with our Specifier- and MethodRef-chains (see subsections 4.3.5 and 4.4.1 respectively). Therefore, we wanted to show the subquery approach here. With Hibernate Criteria, it is impossible to use a subquery for this purpose. This will be discussed in subsection 6.1.4.

On lines 4 and 7 we use two types almost similar to the persistent object types, suffixed with an _. These types are defined by the JPA Metamodel API, where classes suffixed with an _ is the metamodel class corresponding to the original persistent object class. A metamodel class is an actual class, placed in the same package as the original persistent object class. It describes metainformation about the class [26], such as its properties (fields). We can use the metamodel type in the query to refer to properties in the persistent objects. Thus, we achieve a higher degree of type safety than with e.g. Hibernate Criteria. By using the metamodel class we are assured, at compiletime, that the property actually exists and that it has the correct type. That is, the compiler will raise an error if we in the query in listing 6.3 try to compare `name` with anything else than an object of type String (on line 8).

However, we have experienced, during testing, that the compiler sometimes avoids to raise errors about type mismatches or incompatibilities. By creating such errors on purpose we have sometimes ended up with a runtime error instead, for unknown reasons. We have not investigated this issue thoroughly enough to idenfity all possible sources of this error, we only note that it occurs occationally.

The metamodel types can be created manually, or automatically with a tool. Several tools exist, e.g. Hibernate Metamodel Generator [1] and Apache OpenJPA Annotation Processor [2].

The metamodel types introduce additional elements that the user must deal with. While they facilitate necessary functionality in terms of type safety, they also are distractions. In Fluenty, similar functionality is implemented without these additional types.

About Hibernate Criteria, we remarked that since we specified property names using strings, automatic refactoring of our domain objects may invalidate our queries, and errors like missing properties remain undetected until runtime. With the metamodel objects, these problems are eliminated. However, we must re-create the metamodel objects after every change in the domain object, to reflect the changes in the metamodel objects as well.

QueryDSL

```
1 JPAQuery query = new JPAQuery ( entityManager ) ;
2 QBook book = QBook.book;
3 QAuthor author = QAuthor.author;
4 List<Book> books = query.from(book).where(book.authors.contains(
5     new JPASubQuery().from(author).where(author.name.eq("Rune
          Flobakk")).unique(author)))
6     .list(book);
```

Listing 6.4: Query with QueryDSL

The backend has an impact on how a query is written. To make the conditions as similar as possible for each technique, we have used JPA as backend. It assumes that the `entityManager` is instantiated properly. Note

[1] http://www.hibernate.org/subprojects/jpamodelgen.html

[2] http://openjpa.apache.org/docs/latest/ch13s04.html

that JPA, and hence the EntityManager, is used by all four techniques. Thus, they can all be used interchangeably.

Each query need to be initialized (line 1). We see that it is closely bound to the backend, as it uses the current `entityManager` session directly. `JPAQuery` is a specific query type. E.g., if Hibernate is used as backend, a `HibernateQuery` must be instantiated instead.

QueryDSL adds a new datatype for each persistent object type. So, if the application has an object of type `Book`, QueryDSL will add a new type `QBook`. This type is used in the queries instead of the persistent object type. They can be considered as metatypes, and correspond to the JPA 2.0 Metamodel API classes discussed previously. However, in QueryDSL we can instantiate them directly, like on line 2 and 3, without having to create them manually or by an additional tool. Note that the field `name` is referred to directly. Thus, the get-methods are never used, as the metatypes give access only to the domain object's properties, not its methods.

Even if they are generated automatically, with just a single instantiation, the additional metatypes are still distractions. It would have been more convenient if the user could relate directly to the persistent object types without using the substitute meta types.

Changes in the persistent objects, like change of a property name, are not reflected in the metatypes. But, since they are normal Java classes, we can refactor the change there as well. Thus, the change will then be reflected in the queries. These types also let us benefit of the IDE's auto-completion functionality, and we will receive errors at compiletime if we use non-existing types or try to compare incompatible values. Thus, we must give a string as parameter to the `eq` method on line 5.

6.1.4 Functionality

To provide a complete evaluation of the functionality of all techniques compared to SQL is considered outside the scope of this report. Therefore, the discussion will be consentrated around issues central for the task described in subsection 6.1.1.

Fluenty

At the moment, Fluent Q is a POC, providing a very limited subset of SQL's functionality. The subset is so small that further evaluation is considered futile. The description of functionality in section 3.3 on page 19 speaks for itself. Suggestions for future additions and enhancements of Fluenty is described in section 8.2 on page 76.

Hibernate Criteria

To solve the task with JPA 2.0 Criteria query, shown in listing 6.3, we used a subquery. Even if Hibernate Criteria also is a so-called criteria API, it does not provide the necessary expressiveness to support the usage of subqueries for this purpose. Since the association between `Book` and `Author` is unidirectional, we must solve the task by using Hibernate Criteria's equivalent to join. Hibernate Criteria does not provide a dedicated

construct for subqueries. However, a `DetachedCriteria`-instance can be used to express a subquery. A `DetachedCriteria` allows the user to create a query unbound to a specific session (see subsection 2.1.1 for more information about sessions). If it had been a bi-directional association between `Book` and `Author`, we could have used a `DetachedCriteria` as a subquery to solve the task with Hibernate Criteria as well [3].

In listing 6.2 we see the usage of a method, `Restrictions.eq`. The `Restriction`-class contain many other useful methods, like `in`, which can be used to determine whether a specific value is present in a collection of values. To solve the task (see 6.1.1), it would be practical if a query could be used as parameter to such methods. That is, the query that retrieves the collection of values. However, this is impossible, as a query cannot be used as a restriction directly. In situations like this, the queries must be run separately from eachother, and a result from the first can then be used as parameter to `Restrictions.in()`.

JPA 2.0 Criteria

As shown by its name, JPA 2.0 Criteria is also a so-called criteria API, and shares most of its functionality with Hibernate Criteria. But, regarding the task, we have found at least two significant differences. As shown in listing 6.3, we use a JPA 2.0 Criteria `Subquery`. As mentioned, Hibernate Criteria does not support the usage of subqueries here, with a unidirectional association between `Book` and `Author`.

Additionally, JPA 2.0 Criteria gives the opportunity to use queries as restrictions. We see that on line 8 in listing 6.3. We use the subquery as parameter to the method `isMember`, which is JPA 2.0 Criteria's equivalent to Hibernate Criteria's `Restrictions.in`. Thus, JPA 2.0 Criteria provides multiple ways to construct a query that solves our task.

QueryDSL

When querying JPA with QueryDSL, the framework translates the query into a JPQL query. Therefore, we define the functionality and expressiveness of QueryDSL when querying JPA to be equal to JPQL's functionality.

To perform a complete evaluation on all differences between JPQL and SQL is outside the scope of this report. We have used the JPQL Language Reference[24], together with own testing experiences, to point at some key points where JPQL provides a lower degree of functionality than SQL. However, they are not directly related to the solution of our task.

JPQL's list of built-in functions is limited compared to SQL. It has some functions that returns numerics, like length, abs and mod, some for date and time values, and some for returning string values, like substring and concat. Additionally, JPQL supports neither union, intersect, limit nor count(*). It does not allow subqueries in the FROM clause either.

[3] http://stackoverflow.com/questions/8656676/hibernate-criteria-collection-property-subquery

6.1.5 Understandability

We have chosen to focus the evaluation of understandability on how easily the query can be translated into natural English. We are aware that investigating how the queries are interpreted by a group of actual people, possibly with different types of knowledge, would have given more interesting and comprehensive results. However, this proved difficult to conduct within the scope of this master thesis work.

Fluenty

If we translate the query presented in listing 6.1 into natural English we get something like "find all books with authors having a name equal to Rune Flobakk". We see that many of the words appear in both the query and the natural translation. Hence, there is a noticeable relationship between Fluenty and natural language. Observe also that the order the words appear in is relatively similar in both the query and the translation.

The noticeable relationship between the query and natural language is not suprising, since Fluenty uses the principles of fluent interfaces in an extensive manner. According to Fowler and Evans [10], such fluent APIs is designed to be easy readable and to "flow".

The keyword `having` has a different meaning in Fluenty than in SQL. For developers familiar with its meaning in SQL, this can be a possible source of misunderstandings. In Fluenty, it's used to specify which persistent object type, and which method in that object, we are interested in. In SQL, it's used to filter the results returned by a `GROUP BY` clause. However, once aware of it, this difference should not be insurmountable to deal with.

Hibernate Criteria

Since Hibernate Criteria does not utilize a fluent interface, queries are constructed quite differently than with Fluenty. A separate query object, of type `Criteria`, must be instantiated. Modifications and restrictions of the query are done by invoking different methods in the query object. Translation of the query directly into natural language will not provide a meaningful result. Thus, Hibernate Criteria queries have poor understandability according to our definition of the term.

The query could have been chained more together, by e.g. invoking the method `add` directly after `createAlias` on line 3 in listing 6.2. Even if we then had used one of the most important techniques to achieve a fluent syntax, method chaining, true fluency is much more than that. This is also emphasized in [10].

That Hibernate Criteria refrains from facilitating a fluent syntax results in queries with a more traditional flow. With traditional, we mean a flow that is usually seen in most Java programs, with more separation into several instructions and invocation of just one method per instruction. Methods in a fluent API are often not designed for use in isolation, but instead as parts of a longer chain. Thus, methods in such APIs, like Fluenty, might lack self-explanatory names that makes sense on their own. Therefore, reading documentation, like JavaDoc, of fluent APIs might give little

value. For APIs like Hibernate Criteria, that has, according to our experience, more meaningful method names, documentation can be more valuable. Seen from that point of view, documentation can contribute to a more understandable API.

JPA 2.0 Criteria

Similarly to Hibernate Criteria, JPA 2.0 Criteria does not utilize fluent interfaces. Regarding understandability, we have not found any special differences between the two APIs, and the issues described for Hibernate Criteria apply to JPA 2.0 Criteria as well.

QueryDSL

QueryDSL facilitates a more fluent syntax than the two criteria APIs do.

Before the construction of the query can start, the query object, of type `JPAQuery` must be instantiated, as well as the metatypes. After necessary instantiations, a QueryDSL query has a relatively fluent syntax close to natural English. A direct translation into natural language gives something like "from books where books' authors are contained in authors where author's names is equal to Rune Flobakk". We see that many of the words from the natural language translation appear in the query in listing 6.4 as well, while the flow is a bit stuttering and laborious compared to real, natural language. However, even if the queries have a fluent syntax, most method names do still make sense on their own. They do not necessarily need to be part of a longer chain to make sense. Still, they fit well in a fluent syntax. Additionally, many of the method names originate from SQL keywords. Users familiar with SQL will therefore probably find QueryDSL easy understandable.

6.1.6 Result handling

By result handling, we mean how the results returned by a query can be used further in the program code.

Fluenty

As shown in listing 6.1, the results from the query are returned as a native Java collection of type `List`, containing objects of the same type as specified by the first parameter to the method `find`. The list must be parameterized with the same type. If not, it will result in a warning.

As mentioned, a result can be returned as a single object instead of a collection with the method `findSingle` instead of `find`. That object must be of the same type as specified by the `findSingle`'s first parameter.

Both the collection and the single object can be used further in the code just like any other object instantiated in the usual way.

Hibernate Criteria

On line 6 in listing 6.2 we see the declaraion of a `List`. It's beeing assigned the return value from the method `list` in the class `Criteria`. As the name suggests, it returns the results from the query as a Java `List`. If

a single instance is preferred instead, an alternative is to use the method `uniqueResult` instead. It works similarly as Fluenty's `find`.

The difference between Fluenty and Hibernate Criteria is that Fluenty returns the results implicitly. With Hibernate Criteria it is necessary to invoke a method explicitly in order to make the query return its results.

JPA 2.0 Criteria

JPA 2.0 uses a similar approach as Hibernate Criteria. On line 11 in listing 6.3 we see an invocation of the method `getResultList`. Similarly, to return only a single object, the method `getSingleResult` can be used.

QueryDSL

Similarly to the three other techniques, queries in QueryDSL may return either a collection or a single result. On line 6 in listing 6.4 we see that the method `list` is invoked. This method takes the desired collection type as parameter. In this case the desired type is a `List` containing `Book`-objects. QueryDSL uses the metatypes here as well. Therefore, an object of type `QBook` is passed to the `list`-method, in order to retrieve a `List` of type `Book`. The method `uniqueResult` is used instead of `list` if a single result is required.

6.2 RQ 2. Which constructs can be identified as typical for this type of API-design?

Throughout this report we have discussed several approaches, methods, techniques and patterns that all have been used in the development of Fluenty. As an answer to this research question, we will try to summarize those we believe have been the most important.

Registration of method invocations

This is the key part of Fluenty, and all its functionality is essentially dependent on this feature. It is done with proxy objects with corresponding invocation handlers and thread-local variables. This has been thoroughly discussed in chapter 4, subsections 4.3.1, 4.3.2 and 4.3.3.

State and thread safety

The registered data about the method invocations must be stored in with a structure that facilitates further use of it. The key point is thread safety, which has been discussed in subsection 4.3.3 on page 31.

These two aspects, registration of method invocations and storage of those in a thread safe way, resulted in identification of possible general design pattern. This pattern was discussed in subsection 5.1.5 on page 43.

Query construction by means of fluent interfaces

The ideas, principles and techniques behind fluent interfaces lay the foundation for how queries are constructed in Fluenty. A discussion about

fluent interfaces and query construction in Fluenty is contained in sections 3.2.3 and 3.3 on page 19.

6.3 RQ 3. How to create an architecture well suited to be integrated with other existing ORM-tools?

That the users must be able to use Fluenty directly in an environment where JPA 2.0 is already used was defined as a non-functional requirement. They must be able to change between the two query techniques without any difficulties.

Since Fluenty "produces" JPA 2.0 Criteria query objects internally, it can be taken seamlessly into professional projects were JPA 2.0 is used already. At the current stage, as a POC, it is unlikely that this will be relevant. However, if Fluenty reaches a level of a more complete implementation it will be a more relevant issue.

In addition to just support JPA, it was desireable to create an architecture that supported easy integration with other techniques as well.

This has been achieved because of the following:

- Usage of JPA 2.0. Both Hibernate and QueryDSL, among others, are able to interpret both JPA 2.0 annotations and Criteria queries. Thus we get integration for "free", since Fluenty, Hibernate and QueryDSL can be used interchangeably.
- If Fluenty is going to be integrated with an ORM-tool not supporting JPA, we only have to add new implementations of the interface `CriteriaPopulator`. These will translate Fluenty-queries into another "language" than JPA 2.0 Criteria.

Summarized, since all aspects regarding query translation into JPA 2.0 Criteria queries are placed in classes sharing a common interface, it's sufficient to just add new implementations of that interface to enable integration with another existing ORM-tools. Other parts of Fluenty can remain unchanged.

Each known ORM-tool has some sort of entity manager (see subsection 6.1.2). It might be named differently in different tools, but its purpose is generally similar. The entity manager is used when initializing the Fluenty repository (again, see subsection 6.1.2). Based on the type of this entity manager, it can be determined which tool is currently in use. Based on that information, Fluenty will be able to choose the correct implementation of the interface `CriteriaPopulator`.

6.4 RQ 4. In what other areas than this project might proxy objects be a useful contribution?

In subsection 1.5.1 on page 5, the creation of a new solution where at least parts of it can be generalized, was defined to be one of the most

important characteristics of development research. We will therefore describe some other possible scenarios where the functionality regarding invocation handling can be proven as a useful and valuable contribution. This section must be seen in relation to subsections 4.3.1 on page 27 and 4.3.2 on page 29, as they contain background information about the principles that will be discussed here.

6.4.1 Blocking the actual implementation

In some situations it is desireable to block the actual implementation, i.e. prevent it from being executed. This may be done completely or by delegating the execution to another part of the application. The latter is done i Fluenty, where the logic is performed by an invocation handler. The actual get-methods in the persistent objects, used in the queries, are never executed. As mentioned in subsections 4.3.1 on page 27 and 5.1.4 on page 42, an alternative is to let the invocation handler add a new implementation of all or some of the methods in the proxied class.

Blocking the implementation completely can be considered as equal to the principle "No Operation Performed" (NOP or NOOP) in assembly programming[4]. This seems to be a technique rarely used in Java. (We do not have any formal resources for our claim. It is based on our own experiences and discussions about the topic with other developers.) However, that it's rarely used does not mean that it cannot be useful in particular situations.

We have chosen to call such functionality "feature-toggling". It might be desireable to have the opportunity to turn features on or off in an application that has reached the production level. By production level we mean that the application has been adopted by its users. Feature-toggling can be beneficial in batch jobs, where it can be used to e.g. turn particular operations in a long batch job on or off. By using dynamic proxies, the code that invokes different operations in the batch job does not need to know whether the different operations in the job should be executed or not. If a particular operation is turned off, a NOOP-proxy can be used instead of the actual implementation.

As the name implies, a NOOP-proxy effectively does nothing. It just provides an empty implementation of the proxied class.

6.4.2 Refinement of existing logic

Expanding and refining existing logic is a typical area of interest in Aspect Oriented Programming (AOP). AOP is outside the scope of this project and will not be discussed. The suggestions presented below do not intend to be a full-scale AOP-solution found in Java extensions like AspectJ [5]. It should rather be considered as "AOP-like" or "AOP-light".

A commonly used tern within AOP is cross-cutting concerns. If code responsible for a single concern is spread across several classes or class

[4] http://en.wikipedia.org/wiki/NOP

[5] http://www.eclipse.org/aspectj/

families (or packages), the concern is considered to be cross-cutting [31]. Based on studying several AOP tutorials, logging seem to be the classical example of such a concern and is the starting point in almost every tutorial. Logging is a concern which is not specific for a particular class, it is something extra cut across otherwise unrelated objects. With a dynamic proxy and invocation handling it is possible to create completely isolated functionality for logging of method invocations and their parameters, return values, execution time, and more. Using the dynamic proxy, the desired classes can be extended with this logging functionality without littering the code performing the actual program logic in those classes.

Another common cross-cutting concern is transaction management. The invocation handler for a dynamic proxy can be used to initiate a new database transaction before executing the invoked method, and then commit or roll back the transaction when the method has finished executing. As with the logging functionality, this transaction handling can be inserted into the classes where it is needed by creating a proxy of those classes, without littering the program logic. Different strategies are possible for determinating which methods require transaction handling. This may be done by giving such methods particular names by following a specified naming convention, or by using annotations.

Frameworks like Hibernate also take advantage of dynamic proxies and invocation handling. Hibernate uses dynamic proxies to add extra logic into the persistent objects. This extra logic makes them "Hibernate aware". That is, objects know if they are persistent objects or not. If there is a change in such an object, e.g. a field getting a new value, the object itself knows that it must be re-stored in the database next time Hibernate does a commit or flush. Hibernate uses Javassist[6] instead of cglib for this purpose, but the principles remain the same.

6.4.3 Lazy evaluation

As mentioned repeatedly, invocations of get-methods in the persistent objects, used in the queries, is blocked. However, this initial block does not necessarily mean that the method cannot be invoked later. The `MethodRef`-chain, described in chapter 4.3.3 on page 31 works with `Method`-objects from Java Reflection to achieve its functionality. Currently, Fluenty does not keep these `Method`-objects, it just use relevant methods to get relevant data about the intercepted method invocation. However, it does only require minor changes to make the `MethodRef`-chain keep the `Method`-objects as well. A `Method`-object has a method `invoke` that can be used to execute the method associated with that `Method-object`. Therefore, with the `MethodRef`-chain as basis, we can later execute those methods that were originally intercepted and stored in the chain. With this technique, we can achieve lazy evaluation.

With lazy evaluation, the evaluation of an expression is delayed until the answer is actually needed. A synonym is call-by-need. The opposite

[6] http://www.csg.is.titech.ac.jp/~chiba/javassist/

to lazy is eager evaluation. The lazy evaluation technique is a technique used most commonly in functional programming. Java is natively not a functional language, so other languages are usually preferred to do such programming, espescially languages like Scala which support functional programming natively. However, with the techniques we have discussed in this section, and others, a more functional style can be achieved in Java as well. Different Java frameworks facilitating for such a functional style is gaining more and more popularity. They make it possible to create easy readable and reusable code. One example of such a framework is LambdaJ, mentioned in subsection 3.2.3. This framework has been an inspiration for us when designing Fluenty.

6.5 RQ 5. How well did the selected research and development methods suit the project?

Different methods for both the theoretical and practical part of this project have been described previously in this report. A research method called development research was presented in section 1.5 on page 4, while a small subset of Scrum together with TDD was presented as development methodology in subsections 4.1.1 and 4.1.2 on page 24. Since this research question intends to highlight how the methods fit *this* particular project, the evaluation will to a great extent be based on our own experiences during the work.

When describing our research method, we referred to a term "constructive research" as well as a definition of "contributing constructs". A desire of both the developer and project initiator was to create a construct that could serve as an opener to create other practically useful constructs later. Our research method emphasizes both theoretical and practical work. As presented in figure 1.1 on page 5, the three first phases of development research consist of problem analysis, design, and development of a solution. As the figure indicates, the solution serves as basis for an evaluation resulting in general design principles (the theoretical contribution). We mean that this approach helps avoiding that development is done separately from the theoretical part. Instead, they are considered as coherent, equally important parts, serving the same purpose. We consider this to be an appropriate approach for a project like ours, where acquiring knowledge about a problem domain is an important aspect.

Early in the project it was chosen not to stricly follow a complete development methodolgy. Instead, only a small subset of typical methodology elements was used whenever needed. This has previously been discussed in section 4.1.

The developer lacked experience with both the problem domain and API development, and the project initiator did not have any absolute requirements for Fluenty. The purpose of the project was mainly to discover available opportunities, and let Fluenty develop incrementally while more and more knowledge was acquired. For a project with such a purpose and few stakeholders, we considered it a waste of time to

emphasize precise requirement specification. We considered other areas, as acquired knowledge, as more important. Additionally, it had possibly proven difficult to define requirements formally and strict at an early stage.

Both use case modelling and user stories were considered appropriate requirement specification techniques. User stories were chosen. This has previously been discussed in subsection 4.1.1 on page 23. Based on earlier experiences with use cases, and experiences with user stories from this project, strengths and weaknesses of the chosen technique can be summarized as follows:

Use case modelling:

1. Use cases would have given a more detailed and concrete specification.
2. Due to lack of knowledge and a complete picture of the desired outcome (prototype), it would have proven difficult to create a detailed use case model early in the project. A considerable amount of time would have been spent during the work to update and maintain the model.
3. A detailed model gives a solid basis for creating accurate estimates.

User stories:

1. Fast and easy way to identify and specify requirements.
2. Plain and easy understandable for all stakeholders.
3. Proven to be difficult to estimate accurately. However, lack of knowledge has probably had just as much influence as the technique.

We have experienced some situations where it was difficult to create accurate estimates for a task. And generally, the estimates have had few practical benefits. Although this has not caused major delays, utilizing some additional elements from Scrum regarding estimation and progress monitoring would probably have been beneficial.

Based on the experiences from this project, we present the following advice for projects with similar purpose and number of stakeholders:

- Do not follow a development methodology completely, as they might be unnecessarily complicated for such projects. Instead, use only some elements when needed, and create your "own" methodology.
- Your methodology should be "very" agile. Agile approaches handles changes well. Changes will always come when your product's goals are vague and your knowledge is limited.
- Do not feel tempted to make your project a "hobby-project" by skipping development methodologies completely. Focus on elements regarding requirement identification, specification, estimation and progress monitoring, and observe the relationship between them. This will help ensure a continous progess.

With limited knowledge at the start of a project, it is natural that there will be changes during the work. Espescially for a project like this, where its purpose to a large extent is to do just that, acquire knowledge. New, better ways to solve the same problem do often appear when the level of knowledge increases. That has occured frequently in this project, as solutions often have been changed, refactored and re-built. It is believed that this is where the project has benefitted most from test-driven development.

As long as the problem to solve is the same, the desired output from the application should usually not change even if the solution is changed. Thus will the test suites already written remain unchanged. After a change is made in the code there will be immediate feedback showing whether the application still gives the desired output by running the tests. With automated test execution included in the Maven build lifecycle (see subsection 4.2.1 on page 26), all unit tests are executed automatically. This ensures that a change to one part of the program code does not affect any other parts of the application. We have often experienced that a change does just that, and TDD has proven to be a valuable mechanism to detect such errors.

Many times during the development, it has been clear only what a method should do, not how it should do it. Then, writing unit tests has been a practical way to express that. It also defines a clear goal - to make the test pass. Having said that, unit tests can also narrow the continous search for better solutions. We have experienced that a test was developed early in the development, and remained unchanged for a long time. The solution was refactored and changed to be more efficient. Later, it accidentally turned out that the behaviour described by the test was not desireable and the problem did not longer exist. The test had led to a too large focus on the solution of a problem and removed the focus from the problem itself.

Based on the reasoning above and the discussion from section 4.1.2, we believe that test-driven development has been beneficial for this project. From the experiences and lessons learned, we present the following advice for similar later projects:

- Use a build tool, like Maven, with automated execution of unit tests. This will ensure that unit tests are run often and thus newly created errors will be revealed early. Additionally, it will ensure that all tests are run at the same time. This will help to ensure that making one test pass, will not cause another test to fail.

- Be consequent. Always write tests before program code. Otherwise the test will not serve its purpose and there will be uncertainty about whether code is covered by tests or not.

- Don't carve a test in stone. That is, be open for other suggestions and change a test if other ways to do things show up.

Chapter 7

Evaluation

This chapter contains a presentation of how Fluenty has been tested. It also presents an evaluation of to what extent Fluenty meets the requirements that were specified in section 3.1 on page 15.

7.1 Testing

Various types of testing has been performed in different phases and for different purposes throughout the work. The following subsections contain a description on how this has been done and a presentation of the results.

7.1.1 Unit testing

As mentioned repeatedly, test-driven development (TDD) has been important in this project. A small application with a simple domain model has served as basis for the unit tests. This is the same model as presented in figure 6.1 on page 50.

For storage of the domain model, HSQLDB was used as RDBMS. It runs in memory, requiring little configuration and no extra infrastructure. Thus, each unit test can be executed quickly and completely self-contained.

In each test suite testing query functionality, we have two special methods, annotated with JUnit annotations `@Before` and `@After`. As their names suggest, they are executed before and after each unit test. The before-method creates the `EntityManager` and instantiates the `Repository`, but more importantly it fills the database with a suitable number of objects from the domain model. The after-method removes these objects by doing a rollback of the transaction. When we execute our tests we have no guarantee of the order in which they are executed. Therfore, by letting dedicated methods take care of object persistance instead of letting each unit test perform that task, we can keep control of how many objects we have in the database. That is, we can be certain that a query returning all `Book`-objects will return e.g. 10 objects, an can test against that number in the unit test to verify that the query actually returns all desired objects.

Maven facilitates TDD well. It creates a separate folder for the test suites, in src/test/java. Tests placed in this folder are included in the

Maven build lifecycle. Those who eventually will develop Fluenty further may place their tests in separate packages. However, to be integrated with the existing tests, these must be placed in the same folder.

7.1.2 Functional requirements

All functional requirements have been tested, mostly through unit testing. By unit testing methods "high in the hierarchy", i.e. close to the facade (see subsection 5.1.2), it has been possible to test most of the functional aspects. Below, a summary of Fluenty's functional requirements is presented together with a description of the verification of each requirement. That is, how it has been determined whether each requirement is fulfilled.

1. **Retrieve all objects of a specific type — implemented**
   We have written four queries in a separate unit test, where each query returns all objects of each type of objects from our domain model. Each query returned the desired number of objects. This was verified by comparing the size of the collection the query returned to the number of objects we expected returned by the query. For this, we utilized the the JUnit method `assertThat` together with the Hamcrest matcher method `hasSize`. Hamcrest is a library of matchers for building test expressions[1]. If the Hamcrest matcher returns true, it will satisfy the `assertThat`-method which makes the test go green. Additionally, we used the `hasItems`-method to verify that the collection returned by the query also contains the desired objects, and not just matches the desired size.

2. **Retrieve objects of a certain type, with one or more constraints — implemented**
   This requirement was verified by following the same procedure as for the previous requirement. Tests were written to retrieve `Books` and `Authors`. It was considered unnecessary to test for additional types. A pair of queries was written per constraint method, that is, two using `equal` for both types, another two using `greaterThan`, etc. After each query, the results were verified by `assertThat`, `hasSize` and `hasItems`.

3. **Retrieve objects of a certain type, having a collection containing another particular object — implemented**
   Such queries use the method `with` as a constraint method instead of `equal`, `greaterThan`, etc. This method takes the object we want to check whether is contained in the object or not, as parameter. The requirement was verified by two different queries. For the first query, we provided the same object instance that was created in the before-method (with the `@Before`-annotation), as parameter. To the last query, an object retrieved by a separate query was given. This was done to verify that such queries return desired results even if

[1] http://code.google.com/p/hamcrest/

the actual object instances used as parameter to the method `with` are different.

The two queries both retrieve `Books` with a collection containing a particular `Author`-object. Our domain model does not facilitate other variants, since the relationship between `Book` and `Author` is the only one that is one-to-many. Results were verified by the same approach as the two previous requirements, by `assertThat`, `hasSize` and `hasItems`.

4. **Retrieve objects of a certain type, holding a reference to another object, which satisfies one or more constraints — implemented**
   It was not specified whether this requirement applies to collection objects, or single objects, or both. Support for both kinds has been implemented, with queries that have some differences. For collection objects, we must invoke the method `having` once again after the invocation to the get-method that retrieves the collection from the persistent object. For single objects, we can continue with an invocation of the get-method in the "another" object, like `getPublisher().getName()`. The main reason for this difference is some technical aspects when translating these queries into JPA 2.0 Criteria queries. Our separation into `CollectionPropertySpecifier` and `SinglePropertySpecifier` classes, described in subsection 4.4.1 made it neessary. However, we also believe the difference harmonizes well with the fluent syntax.

   The requirement was verified by retrieving `Books` with `Authors` (a collection property) with a particular name, as well as with `Publishers` (a single property) with a particular name. This length of the method chain was defined as the minimum that must be supported (see the listing of the functional requirements in subsection 3.1.1 on page 15). However, support for longer chains was desireable, but not mandatory. We tried to retrieve `Books` with a `Publisher` with an `Editor` with a particular name. Unfortunately, this test failed. Hence, only the minimum requirement is satisfied.

   The results were verified the same way as the other requirements, by `assertThat`, `hasSize` and `hasItems`.

5. **Compatible return type**
   This requirement was verified while testing those we have discussed so far. For each query, we tried to parameterize the collection with another type than returned by the query, e.g. by writing `List<Publisher>` instead of `List<Book>`. This resulted in a compiler error complaining about a type mismatch.

6. **Return a collection or a single result**
   Similar to the previous, this requirement was also tested simultaneously with the other requirements. For each query, both methods `find` and `findSingle`, were used interchangeably. When `find` was used, we measured the collection's size with the earlier mentioned

method `hasSize`. For `findSingle`, we checked whether the returned object was the one we wanted by using the method `is` from the set of Hamcrest matchers.

If we tried to assign the return value from the query to a collection, `List`, if we used `findSingle`, or opposite, we received a compile error about a type mismatch.

7.1.3 Non-functional requirements

Unit tests do generally not cover the non-functional requirements. Thus, the test going green is only valid in a local context and does not give a real and complete impression about what the net effects of code changes are [8].

Automated testing tools exist for testing of these non-functional requirements as well, but we have chosen to not use any in this project. The project initiator did not require formal specifications about whether these requirements (stated in subsection 3.1.2 on page 16) have been met or not. Thus, it was considered unnecessary to spend time on learning automated testing tools which we have no guarantee that would have given us the desired results. Instead, the non-functional requirements have been tested mostly with the following testing techniques:

- Compatibility testing
- Usability testing
- Performance testing

The usability testing has been performed mainly by the developer and the project initiator. We are aware that since they have in-depth knowledge of Fluenty, they are not impartial. Ideally, to achieve more accurate results, the usability testing should have been conducted by independent users. We do however believe that this does not affect our results significantly.

Performance

- No noticeable difference in performance when using Fluenty instead of JPA 2.0 directly

This requirement was verified by measuring the execution time of two unit tests. Each test contained a query that executed the task defined in the test scenario in subsection 6.1.1 on page 50, namely to retrieve all `Books` with an `Author` with a certain name. One test contained a Fluenty query while the other a JPA 2.0 query. The queries was run against a HSQLDB with 6000 records. Of these, 2000 were books, 2000 authors and 2000 publishers. Each persistent object had their fields set with different values. Thus, one `Book` was the desired output result. The test machine had an Intel Core i5 processor, running at 2,40 Ghz, and 6 GB RAM.

Each test was run in cycles of 50 executions, were each cycle was repeated 10 times. Each cycle involved persistance of the 6000 persistent

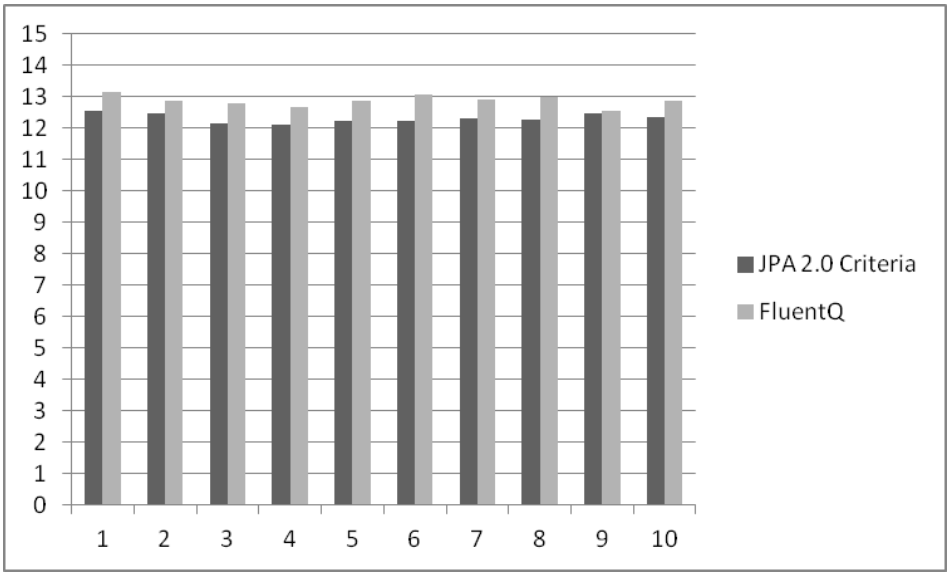

Figure 7.1: Unit test execution times

objects, performed by the method annotated with `@Before` (see subsection 7.1.1).

Figure 7.1 shows the execution times for each cycle. The test containing the JPA 2.0 Criteria query had an average execution time of 12,322 seconds per cycle, while Fluent had 12,880 seconds. Fluentys efficiency is anyway limited to the efficiency of JPA 2.0, since all Fluenty queries is translated in JPA 2.0 Criteria queries, as mentioned repeatedly.

The translation process does not involve traversal of any complex or large data structures or other operations that consumes resources heavily. The bottleneck is probably query optimization of the queries within the `populate`-methods (see section 4.4). All queries are built up within these methods following the same pattern for each query. It is unlikely that this pattern is the most efficient pattern for *every* possible query.

The requirement states that there should be no noticeable difference in performance when using Fluenty instead of JPA 2.0 directly. Our tests has shown that this requirement is not fulfilled. Even if the difference in performance is considered as relatively small, it is still noticeable. In future work, the performance aspects should be taken into consideration. See section 8.2 on page 76 for a discussion around this topic.

Modifiability

- Possibility to easily extend vocabulary and functionality

This requirement is difficult to test formally. Therefore we will present a discussion of our efforts to create an architecture that facilitates modifiability. Since Fluenty's functionality as a POC-implementation is limited, the possibility to easily extend it with new functionality in the future is important. For a better overview, the term "functionality" has been broken up into more specific areas:

1. A wider range of available constraint methods
   Currently supported constraint methods were listed in table 3.1 on page 20. It will probably be a need for a wider range of such methods if Fluenty is to be used professionally. As discussed in subsection 4.4.1 on page 35, all constraint methods have been gathered in the class `SinglePropertySpecifier`. Thus, it's easy to implement more, as this will affect one single class only.

2. Aggregate functions, sorting, and ordering
   Java collections have functionality for many of these aspects. Since Fluenty queries return a Java collection directly, we consider it as likely that the user will find it sufficient to use Java's built-in collection functions when in need of such functionality. However, if this should turn out to be insufficient, extending the class `Where`, shown in the class diagram in figure 5.3 on page 47, will be a natural starting point. This class contains implementations of the method `having` which are utilized when `having` is invoked several times per query, i.e. in queries like `...having(on(Book.class).getAuthors()).having(on(....` We believe that placing methods for such functionality (aggregation etc) here will also make them fit well into the fluent syntax. Imagine replacing the last `having` with e.g. `...getAuthors()).sortBy (Author.class).getAge()..)`.

3. Conditions, like, between
   It might be desireable to have the possibility to query for objects with a field set to a value almost equal to something, e.g. a substring or a pre-/suffix, or between two values. At this point, it is difficult to point at one or several specific classes in Fluenty where methods providing such functionality might fit. However, all classes relevant directly for the query functionality part have been put in one separate package. It was emphasized to try to fill this package with as few classes containing as little code as possible. Hence, it's easier for any future developer to gain an overview over the current functionality and its architecture. It's also easy for him to know where he must place any potential new classes providing new query functionality, in order to integrate them with the rest of Fluenty.

As said numerous times, Fluenty returns objects, either single or in a collection. Hence, it is not possible to e.g. make it retrieve the largest amount of pages in a book, i.e. a number. Any eventual future expansions should not change this, it's original purpose is to create queries to retreieve objects, and we want this purpose to remain unchanged in the future.

- Integration with other ORM-tools

This requirement has the same objetives as research question number 3. A discussion about it was presented in section 6.3 on page 61. Any further discussions or descriptions of this aspect is considered unnecessary.

Testability

- The development should be done with test-driven development

TDD has been used during the entire development process. We have discussed TDD numerous times in several sections, including 4.1.2, 6.5 and 7.1.1, among others, which should serve as sufficient verification of this requirement.

Usability

- Receive feedback at compiletime

Since a Fluenty query is written as native Java code, the developer will receive feedback from the compiler in the IDE if there are errors in the query. This applies to both syntactical errors and type compatibility issues. Examples of such errors were discussed in section 7.1.2.

- Ability to write queries using native Java language with a fluent syntax
- Ready for use in a JPA 2.0 enviroment

We believe both have been argued sufficiently for numerous times throughout this report.

7.2 Limitations

This section presents current identified limitations in Fluenty.

7.2.1 Final classes and methods

The keyword final is used on both classes and methods and variables, meaning that it can not be changed later. Currently, Fluenty does not support this keyword. Two aspects need to be adressed to remove this limitation:

1. JPA 2.0 does currently not allow classes annotated as an entity to be declared final. Neither can methods or instance variables be final. Thus, Fluenty will be unable to support elements declared final, as long as it is built on top of, and bases its functionality on, JPA 2.0. (Unless the JPA specification is changed in future versions.)
2. Technically, an invocation handler is a subclass of the proxied class. A final class can not be subclassed. Thus, it is impossible to create an invocation handler of a final class and thereby also to intercept method invocations to final methods.

Both these aspects have large impact on Fluenty's functionality, and we consider it as problematic to address them. We are currently unaware of any other solution that provides the same functionality and supports final elements at the same time.

7.2.2 Functionality

We stated earlier in this report that the purpose of creating Fluenty not was to create a tool ready for professional use. Thus, Fluenty provides only the most basic functionality necessary to write and execute simple queries against a database.

Even if it was not intended for professional use, it can be considered as a limitation that Fluenty currently is unsuitable for this purpose. Emphasis must be placed on expanding Fluenty's functionality considerably to make it reach a functionality level that will satisfy professional developers. We discussed functionality that might be of particular interest in subsection 7.1.3 on page 70. Additional proposals for further work will be presented in section 8.2.

Chapter 8

Summary

This chapter summarizes the work related to this master thesis project, and proposes some areas for possible further work with Fluenty.

8.1 Conclusion

We can summarize the main goals of this masther thesis project with the following:

- Identification of constructs and patterns relevant when creating DSL-like APIs in Java
- To show how these findings could be used to develop such APIs, a prototype of a fluent, type safe API for queries against relational databases was developed

Requirements to the new API were identified by evaluating a range of current ORM query techniques.

The objective was to create an API where queries could be written type safe. Methods in the persistent objects are accessed directly in a statically typed way. That is, no strings are involved when writing queries, as it's done by using constructs in the programming language (Java). This admits the compiler to check for errors regarding both syntax and type mismatches and incompatibilites.

In addition to offer complete type safety, queries are written by using a concept we have referred to as a fluent interface. This results in queries more readable and understandable. They are constructed with a syntax significantly closer to natural English, in terms of both vocabulary and structure, than traditional Java program syntax.

Dynamic proxy objects with invocation handlers, realized by Cglib, Java Reflection, Java's facility for thread safe variables (ThreadLocal), and Java Persistence API, form the foundation for Fluenty's functionality. The development has followed an agile approach, with focus on test-driven development.

The evaluation of Fluenty in the context of our research questions pointed at both similarities and the most significant differences between

a range of ORM query techniques. We found that Fluenty provides the most fluent syntax and high level of type safety. On the other hand, its functionality is strongly limited compared to the other tools. We pointed at architectural decisions that facilitates easy future development and expansion of Fluenty, as well as some typical constructs for this type of API design.

One important aspect in our research methodology, introduced early in this report, was to create a solution where at least parts of it are generalizable. We used our findings regarding techniques, constructs and patterns to create an API for queries against databases. These general techniques should be suitable when creating APIs for different purposes as well. In that context, we identified and discussed a possible new design pattern. We discussed other usage areas where proxy objects and invocation handling can be a useful contribution. We also provided a discussion, with focus on development, about how well our selected methods suited this project.

The evaluation of Fluenty showed that we have created an API that adheres well to its intitial requirements. However, functional requirements were few, and we believe there is unrealized potential in expanding Fluenty's vocabulary and functionality.

8.2 Further work

Fluenty is currently a functional prototype. It provides some basic functionality, but it needs to be enhanced before it can be used in real-life, professional projects. Additionally, it would have been desireable to look into other technical approaches than those we have used. This section elaborates which areas that might be most interesting to consider for further work.

8.2.1 Support for longer invocation chains

Currently, only chains consisting of two methods is supported, like in the following query:

$$...having(on(Book.class).getPublisher().getName())...$$

It could be desireable to have Fluenty to support chains of an unlimited length, e.g. to support queries like:

$$...having(on(Book.class).getPublisher().getEditor().getAddress().getStreet()....$$

Another example will be:

$$...having(on(Book.class).getAuthors()).having(on(Author.class).$$
$$getAddress().getStreet().getSomethingElse...$$

This should be concidered in context of the topic in the next subsection, as it probably will raise the need for using different approaches than the ones currently used.

8.2.2 Other possible ways to build queries

As described in subsection 4.4 on page 35, queries is currently built by traversing a chain of so-called specifiers, either `SinglePropertySpecifier` or `CollectionPropertySpecifier`. In each specifier, the query is being modified by adding JPA 2.0 Criteria restrictions.

Currently, a chain of only two such specifiers is supported. A chain can consist of one `CollectionPropertySpecifier` followed by one `SinglePropertySpecifier`, or two consecutive `SinglePropertySpecifiers`. Future work will involve either expanding this approach to support a longer chain, or create a different approach to construct queries.

If expanding the current approach is chosen, this will involve the following aspects:

1. Create a structure that supports an unlimited number of both types of specifiers in the chain.

2. Modify the `populate`-method in each specifier.

If a completely different approach is chosen, it will been interesting to see if another technique than JPA 2.0 Criteria could have been used. An unanswered question is: Will this affect the structure and design of Fluenty and its performance?

8.2.3 Richer vocabulary and enhanced functionality

Fluenty's vocabulary needs to be expanded extensively to support more complex queries. This will involve picking out relevant functionality from SQL and transfer it to Fluenty. We have listed some functional aspects that might be useful to look into in subsection 7.1.3 on page 70.

8.2.4 Evaluate the suitability of fluent interfaces for complex queries

As pointed out repeatedly in this report, Fluenty does only support queries with a low level of complexity. If the suggestion in the previous subsection is implemented in Fluenty, one should also analyze whether fluent interfaces still is the most convenient technique to achieve more understandable queries. This evaluation could be consentrated around determination of when, that is, at what level of complexity, does fluent interfaces become unsuitable? Does such a threshold level exist? Does it exist other alternatives, that still maintain the complete type safety?

Appendix A

User stories

As a developer, I want to ...

- write queries using Java
- write queries without having to use any strings (except when I want to compare a string with a value in the entity object, of course)
- be able to refactor my code without having to worry about my queries becoming useless
- have tool support like auto completion, refactoring and such in my IDE when writing queries like I have when I'm writing traditional Java code
- receive feedback in my IDE at compiletime about syntactic errors
- determine direectly when writing the query which methods are available in the entity object
- use the results from the query directly without having to do any casting or similar
- receive feedback if I'm trying to compare incompatible types
- receive feedback if I'm trying to assign the results of a query to an incompatible collection
- change query technique only depending at my own will
- include Fluenty in my programming project by only modifying the Maven POM
- write queries which are understandable also for persons not familiar with SQL
- write queries that needs little documentation
- write queries with a flow that I can translate easily into native English

- add constraints to queries by using methods available in the entity objects. This should be the only option..
- use methods in entity objects referenced from other entity objects, e.g. if a Book has an Author, I want to retrieve Books based on the Authors name

Appendix B

Installation

B.1 Source code

Fluenty's source code is available at my GitHub repository at: https://github.com/roberla/Fluenty.
The code can be navigated and viewed directly online. The entire repository can also be downloaded as a zip-file.

An additional GitHub repository can be found at:
https://github.com/roberla/APIEvaluation.
This repository contains the source code used to evaluate the different query techniques in chapter 6.

B.2 Installation

B.2.1 Prerequisities

The following tools need to be installed in order to try out Fluenty in practice:

- Java (version 1.6 or later)
  http://www.java.com/en/download/manual.jsp
- Maven 2.X/3.X (only 3.X has been verified)
  http://maven.apache.org/download.html

Convenient, but not necessary:

- Git
  http://git-scm.com/download

B.2.2 Downloading, building and running

1. Download the source code. If you installed Git, simply run the command `git clone git://github.com/roberla/Fluenty.git`. Otherwise, download (and extract) the zip from the repository at GitHub (URL provided above).

2. Fluenty's source code is now contained in a folder named `Fluenty`. Navigate to that folder, and execute the command `mvn package`. If everything works, Maven will now build the project, execute the unit tests and give the acknowledgement "BUILD SUCCESS".

3. If you want to import the source code into an IDE (Eclipse), you can do so by the command `mvn eclipse:eclipse`. Maven will then generate necessary Eclipse project files. Fluenty can then be imported into Eclipse as a normal Eclipse project.

Bibliography

[1] Dave Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[2] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33 –40, nov. 1998.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, second edition, 2003.

[4] Christian Bauer and Gavin King. *Hibernate in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[5] Rahul Biswas and Ed Ort. The java persistence api - a simpler programming model for entity persistence. Technical report, Oracle, 2006. http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html.

[6] R.G.G. Cattell. *Object data management : object-oriented and extended relational database systems*. Addison-Wesley, 1991.

[7] M. R. de Villiers. Three approaches as pillars for interpretive information systems research: development research, action research and grounded theory. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT '05, pages 142–151, , Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[8] Kristoffer Dyrkorn and Frank Wathne. Automated testing of non-functional requirements. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 719–720, New York, NY, USA, 2008. ACM.

[9] Rune Flobakk. Automated verification of design adherence in software implementation. Master's thesis, Norwegian University of Science and Technology, 2007.

[10] Martin Fowler. Fluent interface. http://www.martinfowler.com/bliki/FluentInterface.html, 2005.

[11] Martin Fowler. *Domain-specific languages*. Addison-Wesley, 2011.

[12] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Brian Goetz. Java theory and practice: Decorating with dynamic proxies. Technical report, IBM Corporation, 2005. http://www.ibm.com/developerworks/java/library/j-jtp08305/index.html.

[14] A. Gupta and P. Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 285 –294, sept. 2007.

[15] Michi Henning. Api design matters. *Queue*, 5:24–36, May 2007.

[16] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*, pages 36 –43, march 2009.

[17] Brown William J., Malveau Raphael C., Hays W McCormick III, and Thomas J Mowbray. *AntiPatterns. Refactoring Software, Architecture and Projects in Crisis*. John Wiley and Sons, Inc., 1998.

[18] Sutherland J. and Clarke D. Java persistence. http://en.wikibooks.org/wiki/Java_Persistence.

[19] G. King, C. Bauer, M. Rydahl Andersen, E. Bernard, and S. Ebersole. Hibernate Reference Documentation. Technical report, Red Hat Middleware LLC, 2004. http://docs.jboss.org/hibernate/core/3.3/reference/en/pdf/hibernate_reference.pdf.

[20] Jeff Langr. *Agile Java : crafting code with test-driven development*. Prentice Hall, 2005.

[21] Craig Larman. *Applying UML and patterns*. Prentice Hall, third edition, 2005.

[22] E. Miller. Advanced methods in automated software test. In *Software Maintenance, 1990., Proceedings., Conference on*, page 111, nov 1990.

[23] Dave Minter and Jeff Linwood. Querying with HQL and SQL. In *Pro Hibernate 3*, pages 145–158. Apress, 2005.

[24] Oracle. *JPQL Language Reference*. http://docs.oracle.com/cd/E12840_01/wls/docs103/kodo/full/html/ejb3_langref.html.

[25] Oracle. *Java EE 5 Tutorial*, 2010. http://docs.oracle.com/javaee/5/tutorial/doc/javaeetutorial5.pdf.

[26] Pinaki Poddar. Dynamic, typesafe queries in jpa 2.0. Technical report, IBM Corporation, 2009. http://public.dhe.ibm.com/software/dw/java/j-typesafejpa-pdf.pdf.

[27] Chris Richardson. ORM in dynamic languages. *Queue*, 6(3):28–37, 2008.

[28] John Smart Ferguson. Hibernate querying 102: Criteria api. http://www.javalobby.org/articles/hibernatequery102/, 2010.

[29] Jan van den Akker, editor. *Design Approaches and Tools in Education and Training*, chapter 1. Kluwer Academic Publishers, 2002.

[30] Pieter van Zyl, Derrick G. Kourie, Louis Coetzee, and Andrew Boake. The influence of optimisations on the performance of an object relational mapping tool. In *SAICSIT '09: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 150–159, New York, NY, USA, 2009. ACM.

[31] Dong Zhengyan. Aspect oriented programming technology and the strategy of its implementation. In *Intelligence Science and Information Engineering (ISIE), 2011 International Conference on*, pages 457 –460, aug. 2011.