

Automatically Detecting ORM Performance

Anti-Patterns on C# Applications

Tuba Kaya

Master's Thesis

07–09-2015

Master Software Engineering

University of Amsterdam

Supervisors: Dr. Raphael Poss (UvA),

Dr. Giuseppe Procaccianti (VU),

Prof. Dr. Patricia Lago (VU),

Dr. Vadim Zaytsev (UvA)



UNIVERSITY OF AMSTERDAM



Abstract

In today's world, Object Orientation is adopted for application development, while Relational Database Management Systems (RDBMS) are used as default on the database layer. Unlike the applications, RDBMSs are not object oriented. Object Relational Mapping (ORM) tools have been used extensively in the field to address object-relational impedance mismatch problem between these object oriented applications and relational databases.

There is a strong belief in the industry and a few empirical studies which suggest that ORM tools can cause decreases in application performance. In this thesis project ORM performance anti-patterns for C# applications are listed. This list has not been provided by any other study before. Next to that, a design for an ORM tool agnostic framework to automatically detect these anti-patterns on C# applications is presented. An application is developed according to the designed framework. With its implementation of analysis on syntactic and semantic information of C# applications, this application provides a foundation for researchers wishing to work further in this area.

Acknowledgement

I would like to express my gratitude to my supervisor Dr. Raphael Poss for his excellent support through the learning process of this master thesis. Also, I like to thank Dr. Giuseppe Procaccianti and Prof. Patricia Lago for their excellent supervision and for providing me access to the Green Lab at Vrije Universiteit Amsterdam.

Since the beginning, this study has been a wonderful journey and a life changing experience for me. I would like to thank following persons for being such inspiring and supportive teachers:

Drs. Hans Dekkers, Prof. Dr. Jurgen J. Vinju, Dr. Tijs van der Storm, Dr. Vadim Zaytsev – also for his support during the process of this master thesis, Prof. Dr. Hans van Vliet, Prof. Dr. Jan van Eijck and Dr Magiel Bruntink.

Contents

Chapter 1 - Introduction.....	1
Chapter 2 - Literature Review	4
Chapter 3 - ORM Performance Anti-patterns	9
Chapter 4 - ORM Performance Anti-Pattern Detection Framework	14
4.1 Research Methodology	14
4.1.1 Initial Analysis of ORM Tools	15
4.1.2 Designing an Abstract Object Model	19
4.1.3 Validation of Abstract Object Model.....	22
4.2 Proposed Detection Framework	24
4.2.1 Data Extraction.....	25
4.3 Case Studies	34
4.3.1 Analysis of Implementation in Case Studies	35
4.3.2 Analysis of Database Entity Declarations.....	35
4.3.3 Analysis of Code Execution Path Structures.....	36
4.4 Results	38
4.4.1 Analysis of Extracted Data.....	43
4.5 Summary	44
Chapter 5 – Experimentation Design	45
5.1 Environment Setup at Green Lab	45
5.2 Experimentation Design	47
Chapter 6 – Analysis and Conclusions.....	51
6.1 Analysis and Interpretation of Results	51
6.2 Answers to Research Questions.....	52
Chapter 7 - Recommendations for Future Work.....	54
Appendix A - Source Code of Anti-Pattern Detection Framework.....	57
Appendix B - Extracted Data Examples	74

Chapter 1 - Introduction

An empirical study suggests up to 98 % performance improvement can be achieved in response time by refactoring ORM performance anti-patterns [Chen14]. In this study, Chen et al. drew their conclusions by running experiments using only Java applications. Though, to be able to reach a general conclusion on performance impact of refactoring anti-patterns of ORM usage and to know whether the results from the existing study are applicable to other platforms, replication studies in other languages are necessary to be conducted.

According to the TIOBE Index, Java, C++ and C# are the most widely used object oriented languages in 2015 [TIOBE15]. There we can also see that since 2002 C++ language popularity has been decreasing. On the other hand, C# language popularity is on the rise. Given these usage statistics and popularity index between these languages, in this thesis project C# applications are targeted.

In this research project a framework was designed and presented to automatically detect Object Relational Mapping (ORM) performance anti-patterns on C# applications. Due to time limitations, only a part of the designed framework was implemented as an application in C# language.

The designed framework and the developed application stand as a foundation for researchers who are willing to work further in this area to allow replication and extension of the study of Chen et al. so that more general conclusions can be drawn on performance impact of refactoring ORM performance anti-patterns.

1.1 Background Information

Today's applications are developed using high level languages following object orientation principles. However, databases chosen as default to store applications' data are not object oriented but relational. During last decades, most effort has been put into development of these Relational Database Management Systems (RDBMSs) due to the flexibility these database systems offer by allowing users to relate records dynamically based on attribute values. While some research exists towards realization of Object Oriented Database Management Systems [Meyer97] [Dearle10], RDBMSs are currently widely adopted in the industry and there is no sign of them fading away.

Object oriented and relational paradigms differ in the data types used, and the way associations and inheritance are implemented. These differences cause difficulties with conversions between objects in the application and the tables in the database (also called object-relational impedance mismatch). To help developers deal with these difficulties, Object Relational Mapping (ORM) tools have emerged and are widely used. These tools allow easy mapping of database entities to tables and columns in design-time; and in the run-time converting between database entities and SQL queries, caching database entities, tracking changes on these entities and transactions for concurrency management. Many light and rich ORM tools exist for most widely used object oriented languages such as Java, C++ and C#.

Chen et al. suggest that wrong usage of ORM tools (anti-patterns) cause significant decrease in the performance of the applications, and refactoring these anti-patterns may result in up to 98% improvement in response time [Chen14].

1.2 Research Methodology

Before building a framework to automatically detect ORM performance anti-patterns, a list of anti-patterns need to be identified. There is no study in the literature that lists ORM performance anti-patterns specifically. This study aims at conducting a list of these anti-patterns. Though, the presented list is limited to the time constraints of this thesis project.

Providing information explained above, the research questions (RQ's) of this thesis project are:

- RQ1: “Which usages of ORM tools are considered performance anti-patterns in the literature?”
- RQ2: “How can an ORM tool agnostic framework for automatically detecting ORM performance anti-patterns be built for C# applications?”

1.3 Structure of This Thesis

In Chapter 2, background information on the need for ORM; features that ORM tools provide and review of related research on performance impact of using ORM tools are explained. Answer to RQ1 with a list of performance anti-patterns is given in Chapter 3. Then in Chapter 4, the research methodology for development of the framework for automatic detection of these anti-patterns is

discussed in detail. Consequently in the same chapter, the developed automatic detection tool for C# applications is presented together with the technical approaches taken to increase its quality as well as its limitations. In Chapter 5, environment setup and experimentation design are documented. Then in Chapter 6, the general aim of this study and conclusions are discussed. Furthermore, the study is recapped and the findings were concluded in the same chapter. Recommendation for future work are given in Chapter 7. Important parts of source code and examples of extracted data for one of the open-source applications used by this project can be found in Appendix A and B.

Chapter 2 - Literature Review

This chapter gives thorough background information that is necessary to understand Object Relational Mapping (ORM), explains the need for ORM and gives a brief summary of studies that have been conducted on ORM performance anti-patterns.

2.1 Background Information

Software systems require rapid changes. To write applications that are easy to change, customize and adapt, object orientation is emerged and used. While object orientation is used for application development, on the database layer it is not adopted. Instead, Relational Database Management Systems (RDBMS) are used as default to store information in logical tables containing rows and columns.

The reason for using RDBMSs with the relational model being more flexible and easier to use than previous database models is explained by Smith et al. [Smith87]. It is more flexible because inter-record relationships do not have to be pre-defined. The relational join operator allows a user to relate records dynamically based on attribute values. The relational model is easier to use because of its more intuitive metaphor of tabular data and because of fourth generation query languages, which allow a user to pose ad-hoc retrieval requests to the system.

There exists some research and interest towards Object Oriented Database Management Systems (OODBMS). According to a study by Dearle et al. there are reasons why in our time RDBMSs should be critically analyzed and there is need for OODBMSs [Dearle10]. Firstly because of the up-most importance of the management of large bodies of programs and data that are potentially long-lived. We have an increased and increasing need for technologies that support the design, construction, maintenance, and operation of long-lived, concurrently accessed, and potentially large bodies of data and programs. Secondly, the world has changed since 1983 (pre-Web) and database applications are pretty much the norm in the Web space of 2010. These Web based database applications have complex data management needs. As stated on the ICOODB 2008 Web page, "Object databases are the right choice for a certain class of application, to save developers cost and time and help them to

build more feature rich OO applications”. Such applications should be a new driver for the development of OODB systems. A study suggests that OODBMSs would bring big performance and maintainability advantages [Meyer97]. However, a study of Leavitt et al. suggests that relational databases have reduced OO databases’ performance advantage with improved optimizers [Leavitt00]. Two decades ago there were initiatives to create object oriented databases (e.g. Matisse and Versant [Meyer97]). However, then focus on these systems and research have faded away. All in all, when looked at the current usage of databases in the world, RDBMSs are the most used and the default database management systems.

2.2 Need for Object Relational Mapping

When object oriented applications access non-object-oriented databases, there needs to be a conversion between objects in the application and the tables in the database. Kulkarni et al. discuss how deep the differences between object oriented programming and relational databases are. Kulkarni et al. suggests that how information is represented, the data model, is quite different between the two [Kulkarni07]. Modern programming languages define information in the form of objects. Relational databases use rows. Objects have unique identity, as each instance is physically different from another. Rows are identified by primary key values. Objects have references that identify and link instances together. Rows are left intentionally distinct requiring related rows to be tied together loosely using foreign keys. Objects stand alone, existing as long as they are still referenced by another object. Rows exist as elements of tables, vanishing as soon as they are removed. Therefore, the impedance mismatch between object oriented and relational database paradigms (object-relational impedance mismatch) are manifested in several specific differences: inheritance implementation, association implementation, data types.

It is no wonder that applications expected to bridge this gap are difficult to build and maintain. It would certainly simplify the equation to get rid of one side or the other. Yet relational databases provide critical infrastructure for long-term storage and query processing, and modern programming languages are indispensable for agile development and rich computation.

Given these differences between object oriented and relational database paradigms, Cvetković et al. summarize disadvantages of traditional data layer development approach to solve these problems as follows [Cvetković10]:

- Development of advanced data layer mechanisms, such as concurrency control, caching or navigating relationships could be very complex for implementation.
- Frequent database structure changes could cause application errors due to mismatch of application domain objects with latest database structure. Those errors can't be captured at compile-time and therefore require development of complex test mechanisms.
- Developed information systems become significantly dependent on concrete database managing system (DBMS), and porting of such a system to another DBMS is complex and tedious task

Instead of traditional data layer development approach, which deals with all these problems for every project, Object Relational Mapping (ORM) tools have been developed and used for object oriented languages such as Java and C#.

2.3 Features of ORM Tools

To address the difficulties that come with object-relational impedance mismatch, ORM tools firstly help developers map their objects to database. Mapping between objects and tables (mapping inheritance hierarchies) can be done in various ways. These ways are following [Mehta08]:

- A single table per class hierarchy
- A table per concrete class
- A table class to its own table

ORM tools then use this mapping information in the run-time to run database code they generate to perform actions on the data. To optimize performance, ORM tools also implement following techniques [Meuller13] [Schenker11]:

- **Object persistence:** The primary way to handle object persistence is to utilize object serialization. Object serialization is the act of converting a binary object, in its current state, to some sort of data stream that can be used at a later date or moved to an alternate location.
- **Object caching:** object/data caching to improve performance in the persistence layer
- **Dynamic querying:** the ability of ORM tools to generate queries to be run on the persistence layer

- **Lazy loading:** optimize memory utilization of database servers by prioritizing components that need to be loaded into memory when a program is started.
- **Nonintrusive persistence:** means that there is no need to extend or inherit from any function, class, interface, or anything else provider-specific.
- **Deferred execution:** The behavior of running a query only when the result is asked.

These rich features help allow developers to focus on other parts of the development rather than dealing with object relational impedance mismatch. However, wrong usage of these features may cause significant decrease in the performance of the applications. These wrong usages are formulated and listed as performance anti-patterns in the literature as explained in following sections.

2.4 ORM Performance Anti-Patterns

Several studies exist that focus on listing performance anti-patterns concerning different types of software systems [Smitt00] [Smitt03] [Roussey09] [Tribuani14] [Tribuani15], detecting anti-patterns [Moha10] or detecting performance anti-patterns specifically [Cortellessa10], [Parsons04].

However, studies focusing on performance anti-patterns caused specifically by ORM usage don't exist.

Cvetković et al. focus on performance impact of using ORM [Cvetković10]. Their study suggests that, while compared to classical database accessing methods, performance of accessing a database using ORM is lower, the difference is not significant (except for bulk updates). Although this study does not suggest significant performance impact with using ORM, it is not focused on specific ways of ORM usage, which are considered anti-patterns by developers in the field.

Chen et al. focused on two of these usages which are considered ORM performance anti-patterns by developers and conducted a study to automatically detect Excessive Data and One-by-One Processing anti-patterns for Java applications. They measured the impact of refactoring the anti-patterns [Chen14]. The results of this study suggest that fixing performance anti-patterns may improve response time by up to 98% (and on average by 35%).

Findings in a study of Chen et al. inspires more empirical studies to be done, since such a significant improvement in performance is highly beneficial for software systems [Chen14]. Authors also

recommend replication of their study for other platforms and languages such as C#, to be able to draw more general conclusions on the impact of refactoring ORM performance anti-patterns.

Chapter 3 - ORM Performance Anti-patterns

The goal of an ORM tool is to let developers take advantage of the querying on an RDBMS while writing object oriented code. Rich ORM tools are involved in mapping of database entities to tables and columns in design-time, and in the run-time converting between database entities and SQL queries, caching database entities, tracking changes on these entities and transactions for concurrency management. These run-time activities are expected to have an impact on performance of the system.

The research I have done in the literature to answer Research Question 1, has revealed a list of ORM performance anti-patterns. There are seven anti-patterns listed in this study. These are either suggested to be anti-patterns in empirical studies or only the problem was explained and the ORM tool usage that caused the problem was claimed to negatively impact performance of applications.

“Excessive Data” and “One-by-One Processing” anti-patterns were already named and described by Chen et al. in their study [Chen14]. However, the other five (“Excessive Data” anti-pattern. “Slow Bulk Update” “Same Query Execution”, “Unnecessary Object Tracking”, “Duplicate Object Tracking”) are named by me in this study.

Below the list is presented in a template for describing an anti-pattern. This template was inspired by the template for design patterns that was originally used in the book Software Architecture in Practice [Bass13]. In this resource, patterns are described by their name, a context, the problem they solve and the solution by which they solve them. When it comes to anti-patterns listed in this study, the context in which does not differ between them. The context can be described as C# applications which use an ORM tool for accessing a Relational Database Management System. On the other hand, for each anti-pattern, it is necessary to clarify their name, the problem they solve and the supposed solution to fix the problem. The anti-pattern template used in this project therefore has following sections:

- Name: Name of the anti-pattern (either given by author of another study or named by me in this thesis project).
- Problem: Brief explanation of the problem caused by this anti-pattern.
- Supposed Solution: How to fix or refactor the anti-pattern.

Two ORM performance anti-patterns suggested and detected by Chen et al. are called “Excessive Data” and “One by One Processing” [Chen14]:

Name	Excessive Data
Problem	Configuring the ORM tool to do EAGER loading, when related entity is not always used together with the main selected entity causes unnecessary performance loss because of the extra joins that have to be made with the query and the bigger data transfer between the database and the application [Chen14].
Supposed Solution	LAZY loading should be configured to prevent excessive data retrieval.

Table 1: Excessive Data anti-pattern definition

Name	One-by-One Processing (N+1 Problem)
Problem	<p>Given there is a one-to-many or many-to-many relationship between two entities, when iteration is done on both entities and LAZY loading is setup, ORM tools do a separate query to the database for each object [Chen14].</p> <p>Example:</p> <p>Company has a one-to-many relationship with Department.</p> <p>When LAZY loading is enabled, following code would generate a separate query to the One-by-One Processing (N+1 Problem) database to retrieve each department object.</p> <pre> for (Company c:companyList){ for (Department d:c.GetDepartment()){ d.GetDepartmentName(); } } </pre>

Supposed Solution	Depending on the ORM tool, configuration needs to be changed to do EAGER loading instead of one by one gathering of data from the database.
--------------------------	---

Table 2: One-by-one Processing anti-pattern definition

As given above, One-by-One Processing anti-pattern is described as related entities being fetched within an iteration. Next to this, even without an iteration an extra call to the database for gathering related entity also can be considered an anti-pattern. Therefore, it is added to the list below as “Inefficient lazy loading”.

Name	Inefficient Lazy Loading
Problem	If the related data is needed when querying the main entity, configuring the ORM tool to do LAZY loading would be inefficient, since then an extra connection would have to be made to database to get the related object. In this case EAGER configuration should be setup for these entities.
Supposed Solution	EAGER loading should be configured to retrieve data for the related entity to prevent extra connection to the database.

Table 3: Inefficient Lazy Loading anti-pattern definition

Stevica Cvetković, Dragan Janković found in their study [Cvetković10] that bulk updates done using ORM tools in C# applications were a lot less performant than performing these updates using traditional database accessing methods. This is explained by the fact that ORM tools have to perform three steps, unlike one as in traditional database update queries, for doing the update of each entity: 1. retrieve the entity from the database as object in the application, 2. change the value of the object, and 3. access the database to save the changes to the object. Therefore, it is suggested that bulk updates using ORM tools is a performance anti-pattern. I called this “Slow bulk update” anti-pattern:

Name	Slow Bulk Update
-------------	------------------

Problem	Set-based update statements cannot be expressed in its original form by ORM tools. Instead, ORM update query must be simulated in three successive steps including querying for the objects to be updated followed by in-memory properties update and finally submit of changes to the database. When dealing with applications that require large sets of records to be updated at once (bulk updates), ORM performance is significantly worse (mentioned as hundred times worse by [Cvetković10]) than not using an ORM tool.
Supposed Solution	Doing the task by running queries against the database with SqlClient rather than using an ORM tool.

Table 4: Slow Bulk Update anti-pattern definition

Name	Same Query Execution
Problem	ORM tools that support deferred execution let you define the query and it is executed once any result of it is used by a piece of code. If the results of the query will be needed more than once, developer might forget about the deferred execution of the code and cause the same query to be run on the database more than once.
Supposed Solution	Do the execution once and assign results to an object, which can be reused.

Table 5: Same Query Execution anti-pattern definition

Name	Unnecessary Object Tracking
Problem	Many scenarios don't necessitate updating the entities retrieved from the database. Showing a table of Customers on a Web page is one obvious example. In all such cases, it is possible to improve performance by instructing the caching mechanism of an ORM tool not to track the changes to the entities
Supposed	Instruct the ORM tool's caching mechanism not to track changes for a specific

Solution	connection.
-----------------	-------------

Table 6: Unnecessary Object Tracking anti-pattern definition

Name	Duplicate Object Tracking
Problem	When ORM tools are used as default, they create the same object twice so when a change is made on the object and changes are submitted, it can compare the object's state when it was retrieved and the changes made, and then decide whether to commit anything to the db server.
Supposed Solution	Keeping two objects is costly. Instead, with some declarations in entity classes, code can take over the statement whether an object is changed.

Table 7: Duplicate Object Tracking anti-pattern definition

Following chapter discusses the approach taken to automatically detect these anti-patterns for C# applications and presents a framework developed to achieve this goal.

Chapter 4 - ORM Performance Anti-Pattern Detection Framework

This chapter discusses the research methodology chosen to answer Research Question 2, “How can an ORM tool agnostic framework for automatically detecting ORM performance anti-patterns be built for C# applications?” and then represents the part of the framework that was developed as part of this study.

4.1 Research Methodology

As an approach to answering a complex question, RQ2 can be split into simpler subsequent questions and answers to these can be put together to eventually answer the complex RQ2. Following this approach, RQ2 was split into following subsequent questions:

1. How can a framework for automatically detecting ORM performance anti-patterns be built for C# applications?
2. How can this framework be made ORM tool agnostic?

To answer the first question, we need to know which data is interesting to be known about an application to detect ORM performance anti-patterns listed in Chapter 3, and what the technical strategies are to realize extraction and analysis of this data.

Answering second subsequent question requires analysis of available ORM tools and analysis of whether an abstraction can be made between these tools.

Below, results of initial analysis of ORM tools are presented. Then, interesting data to be known about an application to detect ORM performance anti-patterns is analyzed and an attempt is made to create an abstract object model to represent this data. Consequently, the abstract object model is validated against different ORM tools to see whether it can be used for detection of anti-patterns on applications using any of the ORM tools.

This chapter continues to present the developed application to perform automatic detection of anti-patterns. One of the findings of this study is that there are multiple challenges with extraction of interesting data on C# applications. These challenges are explained in detail in section 4.2.1.2. Due to

these challenges and time limitations, I decided to find real life applications and focus on the scenarios which exist on these applications while developing the data extraction libraries. According to criteria I defined, four open-source C# applications were selected. Data extraction on one of these applications was completed and presented at the end of this chapter.

4.1.1 Initial Analysis of ORM Tools

In the industry, ORM tools are categorized into “Light” (also called micro-ORM) and “Rich” ORM tools. The ones that don't provide features other than mapping objects to tables and the conversion between database queries and objects, are put into the group of light ORM tools. Some examples of lightweight ORM tools for .Net framework (C# is an object-oriented language that runs on the .NET Framework) are Dapper, Massive, PetaPoco. These tools allow the developers to write SQL queries in string format, execute the query and convert the result set into objects, classes of which were mapped to tables in the database in a way that the tool requires.

On the other hand, rich ORM tools provide lazy and eager loading, caching and object tracking features. Entity Framework, NHibernate and LINQ to SQL are three of the rich ORM tools for .Net Framework.

In this thesis project, I decided to include only the rich ORM tools in the further analysis. This decision is made because of the fact that performance impact of refactoring anti-patterns caused by wrong usage of features rich ORM tools provide are more interesting to developers in the industry than anti-patterns caused by wrong usage of features light ORM tools provide. This explains why the literature study revealed the list of anti-patterns presented in Chapter 3. Listed ways of ORM usage, which are called anti-patterns within the context of this thesis project and in literature, are commonly expected by developers to cause performance degradation.

LINQ to SQL and Entity Framework are ORM tools built by Microsoft itself. Due to the fact that Microsoft stopped development for LINQ to SQL and continues to develop Entity Framework, for new software systems Entity Framework is more likely to be chosen. Nevertheless, legacy projects might be using LINQ to SQL, since it is a powerful and easier to learn ORM tool when compared with the other two options. NHibernate is an actively contributed open-source ORM tool. It is the .Net version of

Hibernate, which is one of the rich ORM tools used in Java applications.

Language Integrated Query (LINQ, pronounced "link") is a Microsoft .Net Framework component that adds native data querying capabilities to .Net languages. LINQ allows ORM tools to build their own IProvider implementation, which is then called by LINQ classes to communicate with the data source. There are two main purposes of LINQ. Firstly it aims at helping developers write queries in a more understandable, and therefore maintainable way. Without existence of LINQ, developers have to write for example SQL (Simple Query Language) queries while reading data from RDBMSs that use SQL as their query language. When in SQL the queries quickly become harder to read, in LINQ it is found to be easier. Below in Code Sample 1 you can see an example of a slightly hard SQL query and its equivalent in LINQ in Code Sample 2:

```
SELECT TOP 10 UPPER (c1.Name)
FROM Customer c1
WHERE c1.Name LIKE 'A%'
AND c1.ID NOT IN
(
    SELECT TOP 20 c2.ID
    FROM Customer c2
    WHERE c2.Name LIKE 'A%'
    ORDER BY c2.Name )
ORDER BY c1.Name
```

Code Sample 1: SQL query to retrieve customers between 21th and 30th rows, taken from [Albahari15]

```
var query = from c in db.Customers
            where c.Name.StartsWith ("A")
            orderby c.Name
            select c.Name.ToUpper();

var thirdPage = query.Skip(20).Take(10);
```

Code Sample 2: LINQ query to retrieve customers between 21th and 30th rows (in query syntax)

LINQ supports two different syntax of queries to allow developers to choose from the syntax that they find easier to understand. These syntax's are “Query Syntax” and “Method Syntax”. The example given above in Code Sample 2 uses query syntax. Its equivalent in method syntax can be written in following way as shown in Code Sample 3.

```
var query = db.Customers.Where(c=> c.Name.StartsWith ("A"))
                .Select(c=>c.Name.ToUpper()).OrderByAscending();
var thirdPage = query.Skip(20).Take(10);
```

Code Sample 3: LINQ query to retrieve customers between 21th and 30th rows (in method syntax)

Second main purpose of LINQ is to allow writing one query which would continue to work, even if the underlying ORM tool changes. This is achieved through implementation of IProvider interface. ORM tool developers are encouraged to implement a Provider, which implements necessary functionalities to communicate with the data source. Therefore, when LINQ is used in applications on business classes, even if the ORM tool changes (as long as the new tool also has an implementation for IProvider) the queries can stay the same.

LINQ to SQL, Entity Framework and NHibernate all support LINQ. Nevertheless, Entity Framework and NHibernate also have other languages built within to allow querying. These are called Entity SQL and HSQL. While it is more common and preferred approach to use LINQ queries, it is also possible that applications use these other query languages. This adds extra complexity while developing a tool to automatically detect ORM performance anti-patterns. This will be explained together with other challenges faced in section 4.2.1.2.

An initial analysis on these three ORM tools revealed following information as a summary:

- Mapping between classes in the application and the tables in the database are necessary for any of the three ORM tools to be able to do conversion between objects and rows. Classes that are mapped to a table are called “Entities”. LINQ to SQL requires entity classes and properties to be marked with special attributes. Entity Framework supports several ways of mapping. It is possible to declare an entity class by deriving it from a special class, but it is also possible to use POCOs (Plain Old CLR Object) to allow entities to be persistence ignorant. Support for POCOs is also one of the strengths of NHibernate. When it comes to detecting ORM

performance anti-patterns, usage of POCOs adds more complexity to the tool, which will be explained in later sections.

- They all implement client side caching, which is reachable by either accessing a specific class that comes with the library or implementing a new class in the project that derives from a specific class. In the context of this thesis project, this class is called “Data Context”. Data Context manages object identity by logging the object and its corresponding row information in an identity table, when a new object is retrieved from the database. If the object is asked again, no query is done to the database. Instead the object is returned from identity table, which acts as the cache.
- On an instance of Data Context object, select, update or delete operations can be performed. If data wants to be read from the database, a query is written and the query automatically converts the data into an entity. If an entity was updated or deleted, corresponding entity is first retrieved and then a command is given to the Data Context instance to update or delete a given entity. Since multiple entities might have been changed in the cache, the synchronization between entities as objects in the application and the rows in the database are done by calling a save changes method. All three ORM tools track changes on the entities that are stored in the client side cache.
- An ORM tool can be configured in multiple ways, even each tool allows multiple ways within itself, whether to retrieve data for entities that are related in the object model to each other. This functionality is referred to as “Lazy” and “Eager” loading. When eager loading is setup, related entities (entities between which a one-to-many, one-to-one or many-to-many relationship exist) are retrieved together with the main entity which was queried - if the RDBMS uses SQL, then by means of Joins. On the contrary, lazy loading allows the related entities to be gathered by means of making a second connection to the data source when they were used.
- All three tools support “Deferred Execution” by implementing support for IQueryable interface for entity classes. With this functionality, a connection to the database is made only when the data to be retrieved is being used in the code.

These are summary of functionality that LINQ to SQL, Entity Framework and NHibernate support. This information will be referred to from section 4.1.3 while validating the abstract object model, which is presented in the following section.

4.1.2 Designing an Abstract Object Model

To be able to detect the anti-patterns listed in Chapter 3 on C# applications using syntactic and semantic information from design-time, extraction of interesting information needs to be done. To have a detection tool that is agnostic to ORM tool used, the extracted information can better be represented in an abstract object model. This way abstract extractors can be built that work for different ORM tool and this will help create better modularity.

The tool Chen et al. have built to automatically detect “Excessive Data” and “One-by-One Processing” anti-patterns on Java applications [Chen14] follows these steps:

- identification of database accessing method calls in an application,
- generating all code paths that involve any of the database accessing method called identified,
- defining a set of anti-pattern detection rules,
- running the anti-pattern detection rules against the code paths generated.

Inspired by this original study, firstly I decided to make an attempt to identify a logical algorithm as pseudo-code to detect each anti-pattern to be able to better understand the data that needs to be extracted and to better indicate whether extracting design-time information would suffice for detection of listed anti-patterns. During development of the tool it, due to several reasons - such as what the extraction framework supports and need to reduce complexity of scenarios to cover by focusing on case scenarios- these algorithms have been changed. Still, I would like to present the initial algorithms (logical steps) that allowed me to create an abstract object model and have an initial plan for development of the detection tool. Below you can see these initial algorithms for each anti-pattern.

Excessive Data

- Find database entities which are mapped to another database entity to be eagerly fetched

- Find objects of these database entities in data flow graphs
- Look in found code execution paths, whether eagerly fetched entity is used on the database entity object
- List code execution paths that do not use eagerly fetched data as Excessive Data anti-pattern

One-by-One Processing

- Go through all database accessing methods
- Check if the database accessing method is in a loop (for, for-each, while, do while)
- If it is in a loop, list the database accessing method

Inefficient Lazy Loading

- Find database entities which are mapped to another database entity to be lazily fetched
- Find objects of these database entities in code execution paths
- Look in found code execution paths, whether lazily fetched entity is used on the database entity object
- List code execution paths that do use lazily fetched data as Double DB Call anti-pattern

Slow Bulk Update

- Find all loops
- Check whether a database entity object is updated inside the loop
- List database entity objects

Same Query Execution

- Find code execution paths that have more than one database accessing method
- Check whether there are any duplicate database accessing method calls in the code execution path
- If duplicates are found, list the duplicate database accessing method calls

Unnecessary Object Tracking

- Find database entities that are marked to be tracked for changes from already extracted ORM settings
- Find objects for these database entities, which are never updated in any code execution path
- List the objects found

According to the initial algorithms given above, an abstract object model was created that reflects the information that needs to be extracted from a C# application. Below you can see a list of objects in the abstract object model and their definitions:

- **Data Context Declaration:** Represents a class that implements client-side caching for ORM tool
- **Data Context Initialization Statement:** Represents a code line that instantiates a data context object
- **Database Entity Declaration:** Represents an entity class (a class mapped to a table in the database)
- **Database Entity Object:** Represents an object that is of a database entity type
- **Database Query:** Represents a code line which returns IQueryable or its parents and contains a reference to at least one database entity
- **Database Query Variable Declaration:** Represents a code line that declares a variable that is of a database entity type or a collection of database entity type
- **Database Accessing Select Statement:** Represents a code line that invokes a database query or a database query variable or an IQueryable<T> object where T is a database entity type
- **Database Accessing Update Statement:** Represents a code line that invokes update method as the ORM tool requires for updates
- **Database Accessing Add Statement:** Represents a code line that invokes add method as the ORM tool requires for updates
- **Database Accessing Delete Statement:** Represents a code line that invokes delete method as the ORM tool requires for updates
- **Database Entity Object Call Statement:** Represents a code line that calls a property or method of a database entity object, where the property is not type of any database entity and the method is not a LINQ invocation method

- **Database Entity Variable Declaration:** Represents a code line that declares a variable, where the variable is a database entity type
- **Database Entity Variable Update Statement:** Represents a code line that updates any property of a database entity object
- **Database Entity Variable Related Entity Call Statement:** Represents a code line that invokes a property on a database entity variable, where the property type is a database entity type
- **Method Call:** Represents a code line of an invocation either to a database accessing method call or to any other method
- **Execution Path:** Represents ordered list of objects, which can be any of the objects from the object model, as a partial or complete execution path

Then this model was validated against all three ORM tools to see whether it can be used to develop an ORM agnostic tool. Following section validates the model against the three ORM tools.

4.1.3 Validation of Abstract Object Model

From presented abstract object model, the objects that are directly related to ORM tool usage are taken, listed in the table below and for each of them it is explained how they can be detected for one of the three ORM tools.

Object from the model	LINQ to SQL concept	Entity Framework concept	NHibernate concept
Database Entity Declaration	Classes that are marked with <code>TableAttribute</code> .	[Meuller13] Classes that are marked with <code>TableAttribute</code> , or Conceptual model in EDMx file. Or POCOs (Plain Old CLR Object), in which case depending on the setup	Type T in classes which derive from <code>ClassMap<T></code> , or types defined in mapping XML file. [Schenker11]

		of the project an appropriate approach need to be used. (For example they might be derived from a given class that is agreed on by developers to be the base class for entities)	
Database Context Declaration	Classes that are marked with <code>DatabaseAttribute</code> .	Classes that derive from <code>DbContext</code> .	There are no such declarations.
Database Context Initialization Statement	Variables of type <code>DatabaseContext</code> .	Variables of type <code>DbContext</code> .	Variables of type <code>Session</code> (BeginSession () method invocation on a SessionFactory object will instantiate a Session object).
Database Query	LINQ to SQL queries in query or method syntax.	LINQ to Entities or Entity SQL queries.	LINQ to Nhibernate or HQL queries.
Database Accessing Method Call Statement	Invocation expressions performed on database queries (or variables that are assigned with database queries).	Invocation expressions performed on database queries (or variables that are assigned with database queries).	Invocation expressions performed on database queries (or variables that are assigned with database queries).

Table 9: Validation of abstract object model

Table given shows that except NHibernate not requiring declaration of a Database Context, in all other cases ORM tools have a relevant required setup. Also, NHibernate still requires usage of a client side

caching implementation by means of instantiating Session class that comes within the NHibernate library. In the case of NHibernate, while detecting these objects with the detection tool the Session class that comes within NHibernate libraries should be used as the class for Database Context Declaration. This way all information needed can still be extracted.

This information stands as validation for assumption that the given abstract object model can be used to represent information that will be extracted from applications and it is valid for all three ORM tools.

4.2 Proposed Detection Framework

Extraction of information about a given C# application might require design-time or run-time analysis. Due to the fact that Microsoft technologies are commercial, projects which implement a profiler or analyze run-time data are not open-source. Therefore, to extract run-time information automatically for C# applications, a profiler would be necessary to implement. On the other hand, there are Roslyn and NRefactory libraries available to help extract and interpret design-time information.

Given information from Table 9, which analyzes how ORM related data is implemented for the different ORM tools, it is possible that these information can be extracted from design-time information. Therefore, the developed detection tool uses syntactic and semantic analysis on the source code of an application.

Roslyn is a recently developed open-source framework from Microsoft that can generate an AST from a given application using .Net framework. This framework was extensively used in the developed framework. How Roslyn works and its usage is explained in more detail in the following section.

The detection tool is written in C# language. It consists of Presentation and Application layers. Presentation Layer is developed as a Console application. Application Layer consists of abstract object model (Models), extraction libraries (Extractors.EF602, Extractors.LINQtoSQL40,..) and a library (Detector.Main) that illustrates the usage of the right extraction library depending on the ORM tool used by the under analysis application and persists the extracted information in JSON files. Figure 1, below shows the design of modules in the detection tool.

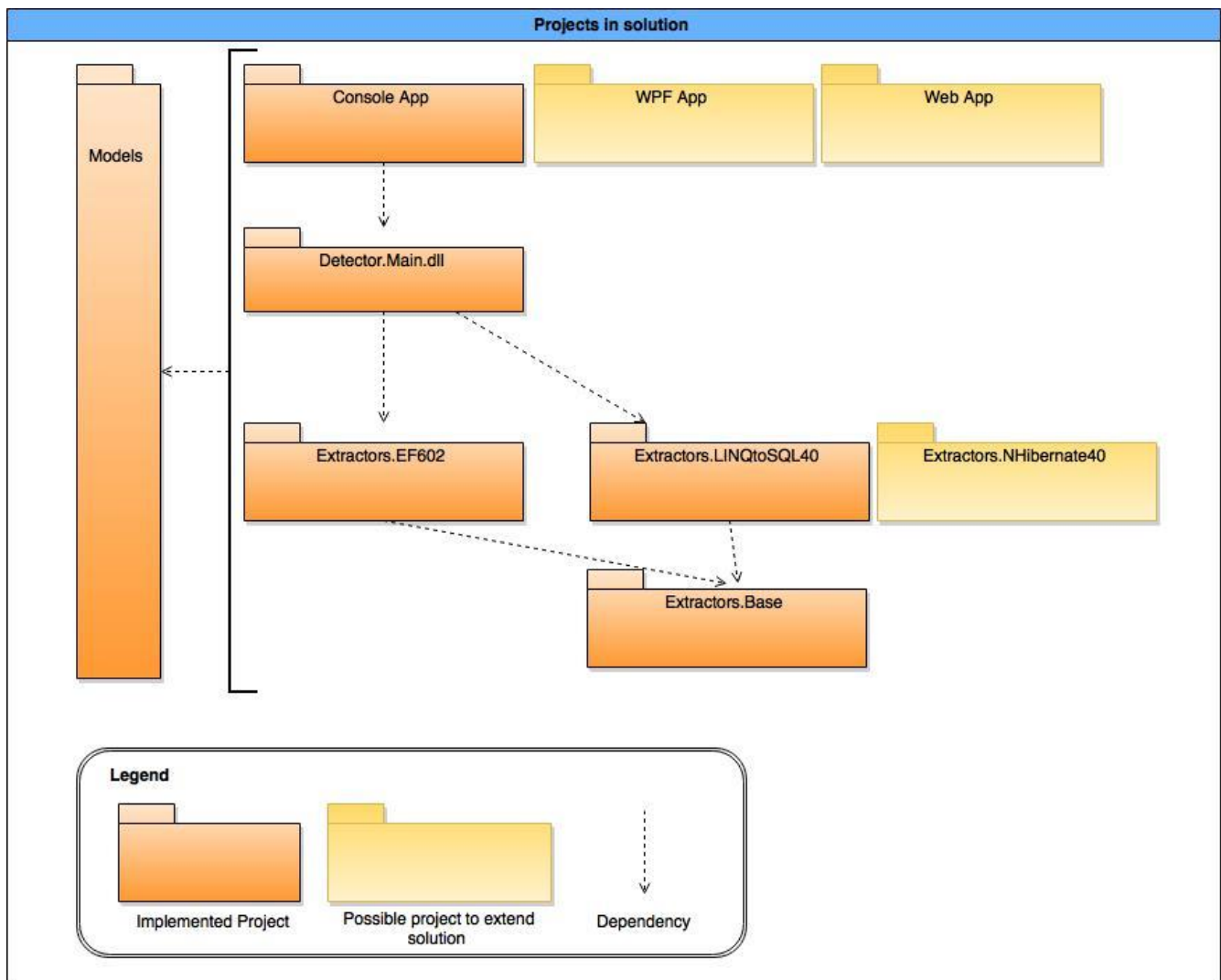


Figure 1: Modules in designed detection framework

Current version of the source code for the automatic anti-pattern detection tool can be found in Appendix A, and it can also be reached on this public GitHub repository.

4.2.1 Data Extraction

The detection tool requires read access to all source code of a C# application. The file path to Visual Studio Solution file for the application needs to be provided, which is used by the tool to access source code. Visual Studio Solution files allow opening projects referenced by that solution file by one Visual Studio instance. It is possible, in real world applications, that projects for an application are spread over multiple solution files by developers to deal with a given set of projects at a time. In this case, to

be able to use the detection tool, a new solution file needs to be created and all projects should be referenced by this solution.

4.2.1.1 Roslyn Framework

Roslyn framework contains a C# parser and extracts the AST for a given C# application. It has three main APIs. These are WorkSpace API, Syntax API and Binding APIs. The tool reaches projects and C# documents in the Visual Studio Solution file by using WorkSpace API. After that, Syntax API and Binding API can be used to reach syntactic and semantic information in each document.

Here, in Code Sample 4, you can see example code using WorkSpace API to iterate through files in all projects of a C# solution file and creating SyntaxTree using the SyntaxAPI.

```
var msWorkspace = MSBuildWorkspace.Create();
var solution = msWorkspace.OpenSolutionAsync(solutionPath).Result;
foreach (var project in solution.Projects)
{
    foreach (var documentId in project.DocumentIds)
    {
        var document = solution.GetDocument(documentId);
        var root = await Task.Run(() => document.GetSyntaxRootAsync());
    }
}
```

Code Sample 4: Usage of Roslyn WorkSpace API

SyntaxAPI uses a model containing SyntaxNode, SyntaxToken and SyntaxTrivia objects to represent a syntax tree. In Figure 2, you can see an image displaying a syntax tree.

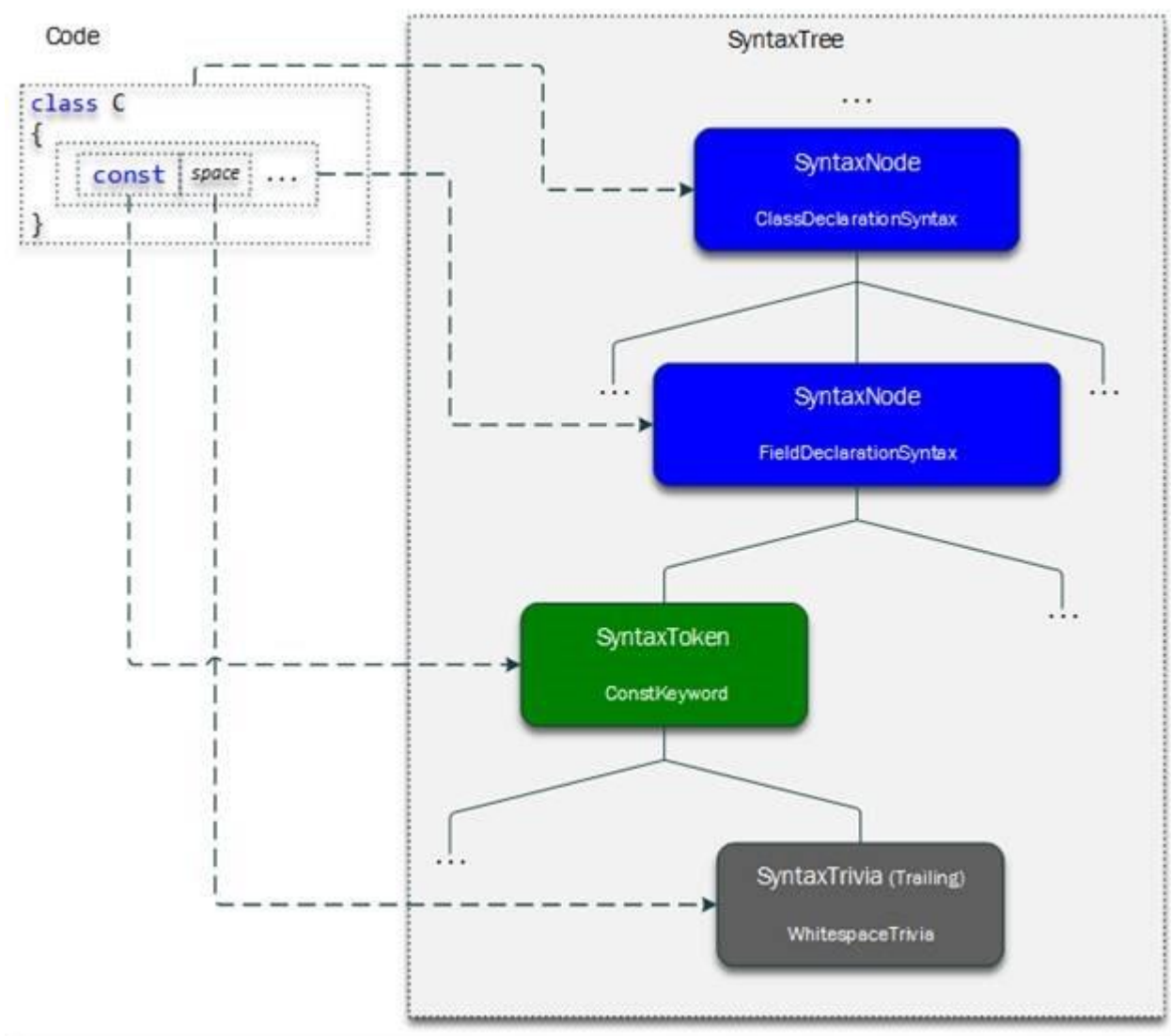


Figure 2: Roslyn SyntaxTree elements

Initially there was no dependency to Roslyn framework from the model classes. However, this choice made it very difficult and time consuming when trying to locate created objects in the syntax tree during the process of detecting objects of the model. For example in one case the only way to find database entity declaration classes is to find the `IQueryable<T>` type properties on data context classes. After data context classes are found, if these objects don't have a reference to the `SyntaxNode` of the class declaration (objects in Syntax API of Roslyn), then the tool must find the `ClassDeclarationSyntax` nodes again. Instead I decided to keep a dependency to Roslyn framework and in a class called `CompilationInfo` to keep the instance of `SyntaxNode` which contained the object

found. This way from any detected object it is very easily possible to refer back to the relevant node in the Syntax API of Roslyn for further analysis.

Here is a class diagram showing the main interface “Model”, class containing Roslyn syntax data for each model “CompilationInfo”, the abstract “ModelBase” class and two example concrete models (DatabaseAccessingMethodCallStatement, DatabaseQuery):

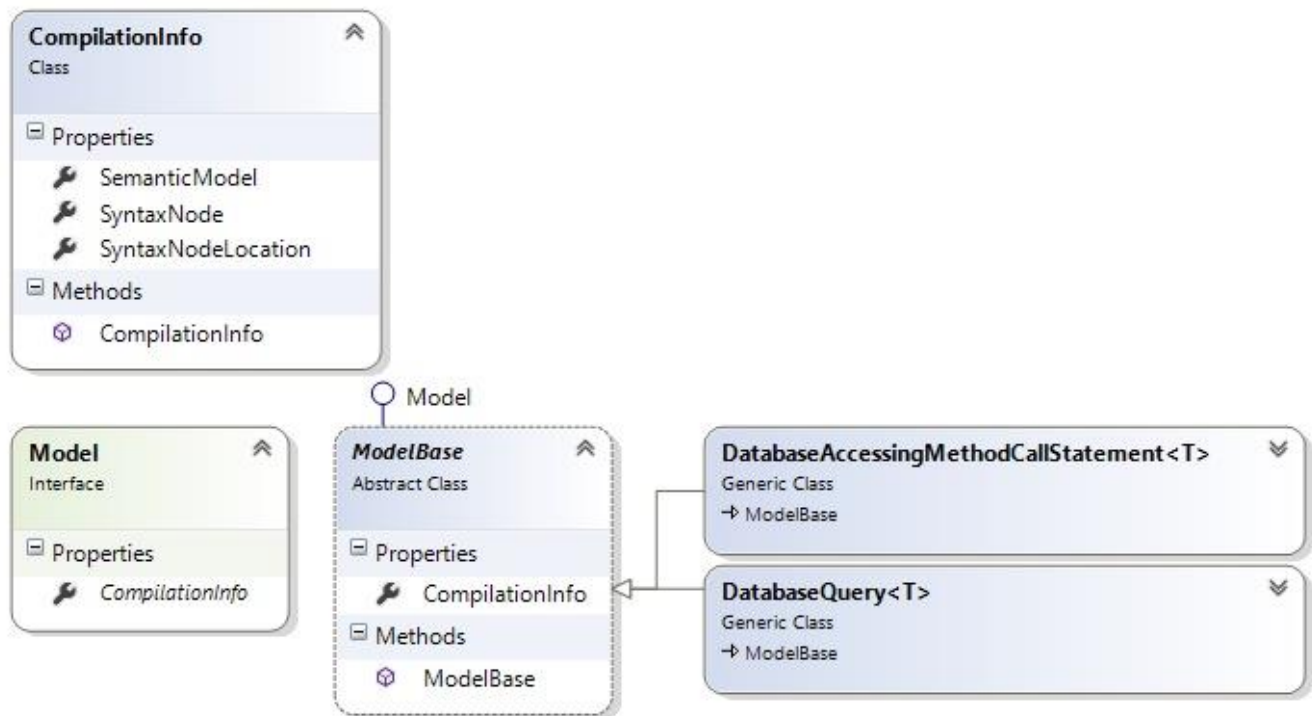


Figure 3: Base Model implementation in detection framework

4.2.1.2 Data Extraction Challenges

Extracting data from C# applications represented by abstract object model given above in <diagram number> is a complex task due to several reasons:

1. There are many ways, in which a code execution path can exist.

A database accessing method call is the line of code which triggers a call to the database.

While using rich ORM tools, with the deferred execution technique, call happens when the results of a query is used. This means that database accessing method calls can be on a line which sets a variable with the value of an object from a query that:

- invokes a method on a query
- triggers an iteration on a query

Another example that shows the complexity of identifying code execution paths is that there can be many ways an entity is used after retrieved by a database accessing method call.

Consider following two examples:

- an entity is returned by a database accessing method call and a related entity is accessed through the entity on a property of a separate class
- an entity is returned by a database accessing method call and it is sent to another method, which reaches the related entity

These are only two possible scenarios given here to explain the complexity of the task in hand. There are many more ways to develop an application while using ORM tools. Given the time limitation of this thesis project, it was not possible to build a tool that can cover all possible scenarios. Therefore, another approach was taken which aims at reducing false negatives (type 1 errors) of generated code execution paths.

2. There are limited frameworks with easy to find documentation available to extract and analyze syntactic and semantic information.

Roslyn framework is a recently released framework with rich capabilities. However, it is not an easy task to perform a real world task with Roslyn. Next to that, there are cases when as a user of the framework one can find herself dealing with unexpected behavior. An example to this situation is that the framework implements two methods with the same signature on classes that inherit from each other, while these methods function differently. This is against Liskov's substitution principle (one of the object-oriented programming principles).

“GetDeclaredSymbol” method declaration on namespaces “Microsoft.CodeAnalysis” and “Microsoft.CodeAnalysis.CSharp” of Roslyn is an example to this problem.

3. Even though syntactic and semantic analysis of source code allows to detect most of the interesting data, there are cases when run-time information is needed.

Finding out whether a select call to the database executes a lazy or eager loading query (in other words whether it loads any related entity's information together with the main entity being queried) is best analyzed from run-time information, which can provide the SQL query to be executed and this query can be analyzed to decide which entities will be fetched by the call. From design-time information, gathering such information is very complicated. There are multiple setup ways ORM tools require to define when lazy or eager loading should be done for a given entity. For example this can be defined on the Data Context Declaration level, Database Entity Declaration level, or Database Accessing Method Call level (in this case it is also possible to do the setup on the initialized Data Context object, which is used by the Database Accessing Method Call). While compile-time data of LINQ does provide the SQL query that will be generated and executed by the providing ORM tool, it has not been possible to perform this task during this thesis project due to its complexity.

4.2.2.3 Extractor Libraries

While the same abstract model can be used to represent extracted data, extractors for each ORM tool require different way to extract data. For each object in the abstract object model, an abstract base Extractor class is implemented in “Detector.Extractors.Base” class library. Then for each targeted different version of ORM tool, a class library is expected to be implemented. These libraries contain concrete Extractor classes which should be derived from mentioned abstract base Extractor classes. Below, Code Sample 5 shows how the base abstract class that all Extractors derive from looks like.

```
public abstract class Extractor<T> where T : ORMToolType
{
    public Context<T> Context { get; }
    public Extractor(Context<T> context)
    {
        Context = context;
    }
}
```

Code Sample 5: Base abstract class for concrete Extractor classes

In this library, a singleton Context class exists. This class is used to store extracted data for a solution. This data is accessed by concrete Extractor classes, when for an ORM tool extraction of different models depend on a sequence. There are cases when extraction of a type of model depends on other already extracted data. Context class is used to provide all extracted data in a given time during execution of the detection tool to extractor classes. The abstract base class for DataContextDeclarationExtractor is shown below in Code Sample 6. Here you can also see that the extracted information is assigned to the related property on Context class.

```
public abstract class DataContextDeclarationExtractor<T> : Extractor<T> where T : ORMToolType
{
    public HashSet<DataContextDeclaration<T>> DataContextDeclarations
        { get; protected set; }

    public DataContextDeclarationExtractor(Context<T> context)
        : base(context)
    {
        DataContextDeclarations = new HashSet<DataContextDeclaration<T>>();
    }

    public async Task FindDataContextDeclarationsAsync(Solution solution,
        IProgress<ExtractionProgress> progress)
    {
        await ExtractDataContextDeclarationsAsync(solution, progress);

        Context.DataContextDeclarations = DataContextDeclarations;
    }

    protected abstract Task ExtractDataContextDeclarationsAsync(Solution solution,
        IProgress<ExtractionProgress> progress);
}
```

Code Sample 6: Abstract base class for concrete DataContextDeclarationExtractor classes

Then a concrete class in for example library “Detector.Extractors.EF602” is implemented for Entity Framework Data Context Declaration extraction. Below in Code Sample 7, the code for this class can be seen.

```
public class DataContextDeclarationExtractor :
    DataContextDeclarationExtractor<Detector.Models.ORM.ORMTools.EntityFramework>
{
    public DataContextDeclarationExtractor(Context<EntityFramework> context)
        : base(context)
    { }

    protected override async Task ExtractDataContextDeclarationsAsync(Solution solution,
                                                                        IProgress<ExtractionProgress> progress)
    {
        string extractionNote = "Extracting Data Context Declarations by finding classes of
                                type DbContext";
        progress.Report(new ExtractionProgress(extractionNote));

        Dictionary<ClassDeclarationSyntax,SemanticModel> classes = await
            solution.GetClassesOfType<DbContext>();

        foreach (var item in classes.Keys)
        {
            DataContextDeclarations.Add(new
                DataContextDeclaration<EntityFramework>(item.Identifier.ToString(),
                                                            item.GetCompilationInfo(classes[item])));
        }
    }
}
```

```
}
```

Code Sample 7: Concrete DataContextDeclarationExtractor class for Entity Framework 6.02

Data Context Declarations in Entity Framework must be classes that derive from DbContext class of Entity Framework. The fact that this is the only way to define them, makes it possible for above class to be sufficient to detect Data Context Declaration objects. However, when it comes to extracting Database Entity Declaration objects and generation of code execution paths, there is no one specific way to follow. Below the approach taken for each case are explained.

4.2.2.4 Extracting Database Entity Declarations

Entity Framework and NHibernate support POCOs (Plain Old CLR Object) to allow persistent ignorance for entities. If POCOs are used in an application, to be able to detect Database Entity Declarations, the tool needs to be aware of the structure of an application. For example developers might have decided to derive all entity classes from a specific abstract class or interface. In this case, the tool must be configured to look for classes with this specific type to extract Database Entity Declarations.

4.2.2.5 Constructing Code Execution Paths

At the beginning of developing the detection framework, extractor classes were written to detect the interesting data with the assumption that all would exist within a given method. In real world, it is seldom the case to find a project that implements such methods. Instead, these data reside in multiple classes and anti-patterns exist across these classes. Therefore, to be able to detect real world cases, it is necessary to first construct code execution paths in a solution. Then for each code execution path, these data need to be extracted and only then detection of anti-patterns can be done properly.

To be able to construct code execution paths, a literature study was done. What I could find was several studies with suggestions on call graph construction algorithms for object oriented languages. These algorithms (RA, CHA, XTA) as compared in [Tip00] differ in the amount of sets they use to construct the call graph and the precision of their results. The main problem that is tried to be solved with these algorithms is to identify which objects would be called as a result of dynamic dispatch in object-oriented languages. Even though these algorithm suggestions exist, constructing call graphs for

an application written in an object-oriented language such as C# is a difficult task. It is difficult because 1) there are many scenarios to be captured, 2) there are close to none existing similar projects for C# applications, which could be used as baseline, 3) usage of Roslyn framework is necessary and there is not much documentation on capabilities of Roslyn. Therefore and given the limited time for completing this thesis project, it is inevitable to have limitations on the cases that can be captured with the detection tool developed.

4.3 Case Studies

Due to complexity explained in previous chapter about the possible ways to define Database Entity Declarations and structures of code execution paths, applications listed below are used as case studies. It is this way aimed to cover the scenario's that exist in these application as a start, therefore to reduce the amount of false negatives.

While choosing these applications, following criteria have been used:

1. Using a relational database to store data
2. Using one of the three most common rich ORM tools for C# applications, which are also the focus for this thesis project (Entity Framework, LINQ to SQL, NHibernate)
3. Being actively used in production (meaning that there is at least one project being implemented using the chosen application, which can be found as a product on internet and that this project has at least one customer)

Due to the fact that Microsoft has limited support for open-source community, it is not common to find many open-source C# applications. However, after some research via the open-source projects supported by Microsoft mainly through [Galloway15], I was able to find four projects which comply the criteria. In table below you can find names of these projects, links to their websites, links to their source code, and the ORM tool each of them use.

Project Name	Links	ORM Tool used
Virto Commerce	Website: www.virtocommerce.com	Entity Framework 6.0.0.0

	Source code: https://github.com/VirtoCommerce/vc-community	
Nop Commerce	Website: www.nopcommerce.com Source code: www.nopcommerce.com/downloads.aspx	Entity Framework 6.0.0.0
Better CMS	Website: www.bettercms.com Source code: https://github.com/devbridge/BetterCMS	NHibernate 3.3.1.4000
Orchard	Website: www.orchardproject.net Source code: https://github.com/OrchardCMS/Orchard	NHibernate 3.3.1.4000

Table 10: Chosen Open-source projects

4.3.1 Analysis of Implementation in Case Studies

Selected applications use Entity Framework and NHibernate. There was no open-source C# application using LINQ to SQL. This might be because development of LINQ to SQL was stopped by Microsoft and therefore it supports less functionality when compared to Entity Framework and NHibernate, as given earlier in section 4.1.1.

These projects all make use of POCO support of the ORM tool. As explained in previous section, in this case to be able to detect Database Entity Declarations, we need to know in which way these classes can be detected.

4.3.2 Analysis of Database Entity Declarations

A manual check on the applications reveal that the way database entity declarations can be detected for these applications are indeed different. In NopCommerce application database entity declaration classes derive from a base class, also implemented within the same solution, called “BaseEntity”. However, in VirtoCommerce application these classes don't inherit from any base class or interface.

On classes that are declarations for data contexts in this application, there are generic collections the type of which are either `IQueryable<T>` or a type derived from `IQueryable<T>` where T are entity type. Therefore, the only way to detect database entity declarations for VirtoCommerce is to first detect these collections and then to find out which class type T refers to.

4.3.3 Analysis of Code Execution Path Structures

On both VirtoCommerce and NopCommerce applications, interesting code execution paths contain calls between two layers/classes. There are Repository classes which contain database queries and database accessing method calls. And some other classes call the methods from Repositories and use returned entities within the same method, if a select was done. Below, in Code Sample 8, you can see example code from VirtoCommerce application.

```
public class PlatformRepository : EFRepositoryBase, IPlatformRepository
{
    ...
    ...
    public AccountEntity GetAccountByName(string userName, UserDetails detailsLevel)
    {
        var query = Accounts;

        if (detailsLevel == UserDetails.Full)
        {
            query = query
                .Include(a => a.RoleAssignments.Select(ra =>
                    ra.Role.RolePermissions.Select(rp => rp.Permission)))
                .Include(a => a.ApiAccounts);
        }

        return query.FirstOrDefault(a => a.UserName == userName);
    }
}
```



```

    }
}

```

Code Sample 8: A Repository class in VirtoCommerce application

Repository classes derive from IRepository interface or a child interface of IRepository, as shown in Code Sample 9.

```

public interface IPlatformRepository : IRepository
{
    IQueryable<SettingEntity> Settings { get; }
    IQueryable<AccountEntity> Accounts { get; }
    IQueryable<ApiAccountEntity> ApiAccounts { get; }
    IQueryable<RoleEntity> Roles { get; }
    IQueryable<PermissionEntity> Permissions { get; }
    IQueryable<RoleAssignmentEntity> RoleAssignments { get; }
    IQueryable<RolePermissionEntity> RolePermissions { get; }
    IQueryable<OperationLogEntity> OperationLogs { get; }
    IQueryable<NotificationEntity> Notifications { get; }
    IQueryable<NotificationTemplateEntity> NotificationTemplates { get; }

    AccountEntity GetAccountByName(string userName, UserDetails detailsLevel);
    NotificationTemplateEntity GetNotificationTemplateByNotification(string
        notificationTypeId, string objectId, string objectTypeId, string language);
}

```

Code Sample 9: Interface for a Repository class in VirtoCommerce Application

Other classes in different projects within the same solution call the method declaration in IRepository interfaces, then use the returned entity as shown in Code Sample 10.

```

public class SecurityController : ApiController
{

```

```

    ...
    ...
    public async Task<IHttpActionResult> DeleteAsync([FromUri] string[] names)
    {
        ...
        ...
        using (var repository = _platformRepository())
        {
            var account = repository.GetAccountByName(name,
                                                    UserDetails.Reduced);

            if (account != null)
            {
                repository.Remove(account);
                repository.UnitOfWork.Commit();
            }
        }
    }
}

```

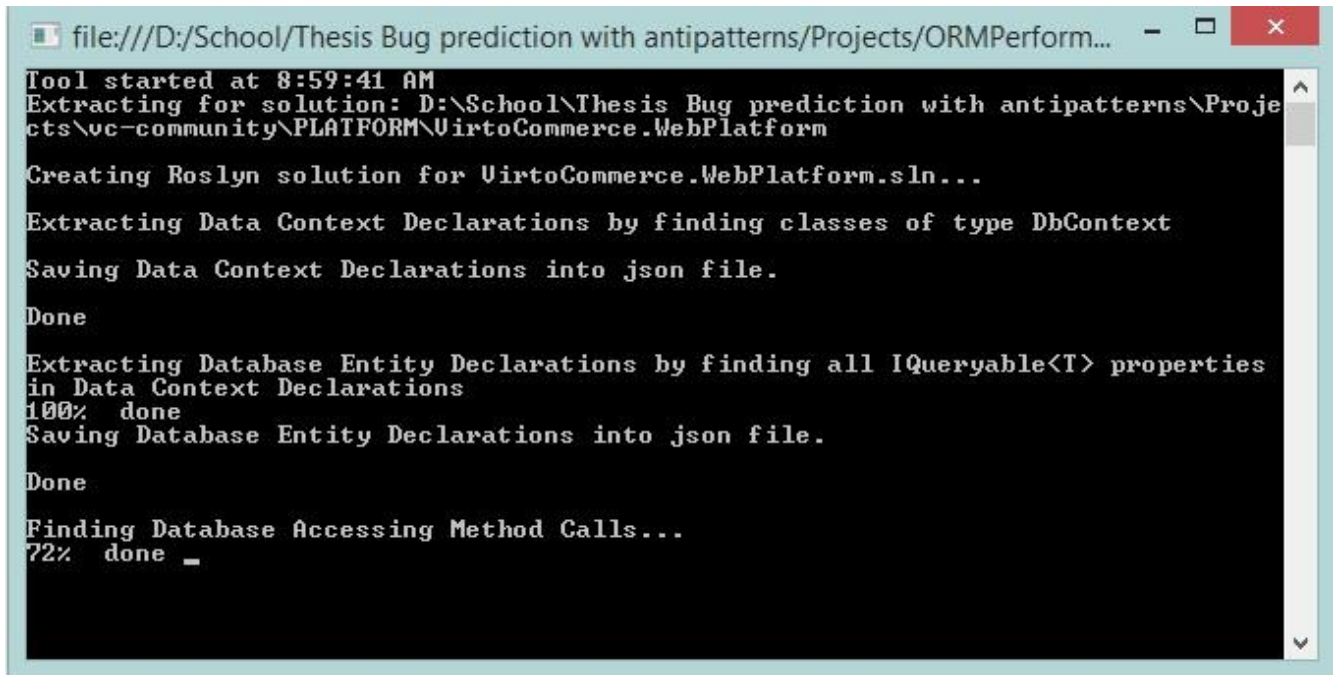
Code Sample 10: A class using a Repository in VirtoCommerce application

Code from the detection framework to extract the Data Context Declarations as example can be found in Appendix A. The rest of the source code is available on the earlier referred to public repository.

4.4 Results

Currently available functionality are extraction of Data Context Declarations, Database Entity Declarations, Database Accessing Method Calls and generation of Code Execution Paths. Time it takes to execute these functionality for VirtoCommerce.WebPlatform solution, which consists of 41462 Lines of Code on 1249 number of C# files, varies in different runs. Because of the variances in execution completion times, the performance of the application was measured three times by running

the application against one solution of the open-source applications, VirtoCommerce.WebPlatform. Below you can see a screenshot of the console application during execution.



```
file:///D:/School/Thesis Bug prediction with antipatterns/Projects/ORMPerform...
Tool started at 8:59:41 AM
Extracting for solution: D:\School\Thesis Bug prediction with antipatterns\Proje
cts\vc-community\PLATFORM\VirtoCommerce.WebPlatform
Creating Roslyn solution for VirtoCommerce.WebPlatform.sln...
Extracting Data Context Declarations by finding classes of type DbContext
Saving Data Context Declarations into json file.
Done
Extracting Database Entity Declarations by finding all IQueryable<T> properties
in Data Context Declarations
100% done
Saving Database Entity Declarations into json file.
Done
Finding Database Accessing Method Calls...
72% done _
```

Figure 4: Snapshot of detection tool's console application during execution

The lowest time it took to complete application execution was 18 minutes and the highest was 25 minutes on the same computer with an Intel Core i7-5500U 2.40GHz CPU with 2 cores.

All extraction functionality are executed as asynchronous methods. Therefore, on a computer with a CPU that has more cores, the tool is expected to complete the execution in a shorter time.

To improve performance of the detection tool further, I have done literature study to find ways to improve usage of data structures. The important point while choosing the right structure for a given operation is to look at the performance requirements for inserting to a set and reading from a set [Mitchell15]. Data being extracted should be placed in appropriate data structures which would result in optimal performance for the tool while performing following operations:

- Operation 1: Insertion of extracted data into the selected data structure for optimized performance of the tool while preparing the structures before detecting anti-patterns on them
- Operation 2: Detecting anti-patterns using the extracted data

To choose the right data structure for operation 1, available collections in C# language were studied. Given that an extracted data set contains only unique items and the order in which they reside in the collection is not of any importance during pure extraction of data, the generic HashSet collection was chosen. HashSet is a collection that contains no duplicate elements, and elements are in no particular order.

Looking at operation 2, to detect each anti-pattern we need to reach a Database Accessing Method and then check existence of models before and after this method in a code execution path. A binary search tree with following structure would give us the best performance while detecting an anti-pattern:

- a root node containing database accessing method as the node object
- on left side the nodes containing models that occur before the database accessing method
- on right hand side the nodes containing models that occur after the database accessing method

While this analysis was done, due to time limitations this optimization is not yet implemented in the detection tool. It is given as one of the recommended future improvements in Chapter 7.

The current implementation done for detecting anti-patterns on extracted data (operation 2), uses a generic IEnumerable collection from C# to store models for each Code Execution Path. This implementation looks like following (Code Sample 11):

```
public class CodeExecutionPath
{
    List<Model> _models;
    public IEnumerable<Model> Models
    {
        get
        {
            return _models;
        }
    }
}
```

```

    }

    public CodeExecutionPath()
    {
        _models = new List<Model>();
    }

    public void AddModel(Model model)
    {
        _models.Add(model);
    }
}

```

Code Sample 11: Code Execution Path model in detection framework

Then classes called Detection Rule implement algorithms to analyze a given Code Execution Path for the existence of a pattern\anti-pattern. An example method analyzing One-by-One Processing anti-pattern is shown in Code Sample 12:

```

public bool PathHasLazyLoadingDbAccessingMethodCallAndRelatedEntitiesAreCalled()
{
    IEnumerable<DatabaseAccessingMethodCallStatement<T>> databaseAccessingMethodCalls
=
CodeExecutionPath.Models.OfType<DatabaseAccessingMethodCallStatement<T>>();

    IEnumerable<DatabaseEntityVariableRelatedEntityCallStatement<T>>
        databaseEntityVariableRelatedEntityCalls =
        CodeExecutionPath.Models
            .OfType<DatabaseEntityVariableRelatedEntityCallStatement<T>>();

    foreach (var dbAccessingMethodCall in databaseAccessingMethodCalls)

```

```

{
    if (!dbAccessingMethodCall.DoesEagerLoad)
    {
        if (databaseEntityVariableRelatedEntityCalls.Any(x =>
            x.CalledDatabaseEntityVariable == dbAccessingMethodCall.AssignedVariable))
        {
            this.DetectedAntiPatterns.Add(new
                OneByOneProcessingAntiPattern(CodeExecutionPath, dbAccessingMethodCall));

            return true;
        }
    }
}

return false;
}

```

Code Sample 12: Method detecting One-by-One Processing anti-pattern on a Code Execution Path

It is recommended as future work to refactor this detection algorithm to use a binary search tree instead of an IEnumerable. Together with this change, the algorithm in given method and in methods for detection of other anti-patterns can be optimized.

Another important point that has significant impact on the performance of the tool is the usage of Roslyn framework. Initially, it takes almost one minute to load VirtoCommerce.WebPlatform solution in memory as Roslyn solution. Next to that, SyntaxNode objects used from this framework are not serializable. The tool could be made more efficient, by serializing and deserializing (part of or all) earlier extracted data for a solution. Currently this is not a possibility, which means the tool extracts all data each time the tool is executed.

4.4.1 Analysis of Extracted Data

This execution results in four JSON files each containing the extracted model information. Following are the model types that are extracted and numbers of objects extracted for each:

- Data Context Declarations extracted: 15
- Database Entity Declarations extracted: 54
- Database Accessing Method Call Statements extracted: 2246
- Code Execution Paths generated: 362

JSON files containing extracted data can be reached on the [public repository](#) of the tool under folder CaseStudyResults. Also examples of elements in these files are presented and explained in Appendix B, at the end of this thesis documentation.

The extraction algorithms were validated firstly by unit testing. Examples of such unit tests are given in Appendix A. This validation and manual analysis of extracted data shows that there are no false positives in extracted Data Context Declarations, Database Entity Declarations and Code Execution Paths.

However, extracted Database Accessing Method Call Statements are currently difficult to validate. This is because Database Accessing Method Call Statement can be any of the following objects:

- Database Accessing Select Statement
- Database Accessing Add Statement
- Database Accessing Update Statement
- Database Accessing Delete Statement
- Database Entity Object Call Statement
- Database Entity Object Update Statement
- Database Entity Variable Related Entity Call Statement

Further development of the tool to categorize extracted Database Accessing Method Call Statements into these objects will make it possible to separately validate extracted data per model. This is another

point of attention which is listed in Chapter 7 as recommended future work.

4.5 Summary

To answer RQ2, common ORM tools for C# applications were analyzed and three most commonly used rich ORM tools were decided to be included in this study. According to the commonalities between these ORM tools, and the data that needs to be known about an application to detect ORM performance anti-patterns, an abstract object model was designed and presented in section 4.1.2.

A framework was designed and implemented that firstly extracts data from a C# application, which are represented by the abstract object model. Roslyn framework was used for syntactic and semantic analysis of code while extracting data.

Several complexities and challenges were faced during the process. These were explained above in section 4.2.1.2. Then, to better understand real life scenario's, four open-source C# applications were analyzed. Results of the analysis were used to adjust the framework to focus on real life scenarios.

Execution of the developed tool for one of the open-source applications resulted in a list of extracted data and generated code execution paths. Performance of the tool was analyzed for this execution and it was explained how detection rules will be used to detect anti-patterns using the extracted data in section 4.4.1. The work that has been done is recapped and contributions of this study are discussed in further detail in Chapter 6.

Chapter 5 – Experimentation Design

During the time for this thesis project, I have setup an environment at the Green Lab of Vrije Universiteit van Amsterdam, which can be used to analyze performance and energy efficiency of applications with and without detected anti-patterns.

5.1 Environment Setup at Green Lab

There are multiple servers at Green Lab on which also a virtual network exists. The servers are connected to physical tools which can measure energy efficiency on separate hardware units such as CPU or RAM.

Before setting up the virtual servers for the experiment, below you can see a table of projects & tools that need to be deployed or ran on the servers to measure performance impact of refactoring the detected anti-patterns, and the software & hardware requirements for each:

Project/Tool to run or deploy	More information on project/tool	Software & hardware requirements
Run open source web application(s)	Open source projects: BetterCMS, Orchard. These projects use Microsoft SQL database	<ul style="list-style-type: none">• For what is known:• Windows Vista SP2 or above ... or Windows Server 2008 SP1 or above (taken from this reference) with IIS 7.0 or above (taken from this reference)• Microsoft SQL Server Express... or if 10GB data storage won't be enough when checking performance

		<p>impact after loading the database, then MS SQL Server 2005 or above[</p> <ul style="list-style-type: none"> • (For ok performance around) CPU: 2 GHz Memory: 2 GB Disk space: 40GB
Run the anti-pattern detection tool	The tool is a Console Application, which needs access to source code of the projects under test	<ul style="list-style-type: none"> • Same hardware and software for running open source web applications can be used to run this tool on.
Run performance test tool(s) and profiler(s)	<ul style="list-style-type: none"> • Visual Studio Performance Testing and Profiling, • PerfView • In case of running into problems with tools listed above, more tools might be necessary to look at 	<ul style="list-style-type: none"> • Taken from this reference & • assuming that < 500 virtual users will suffice for this experiment: • Minimum CPU: 2.6GHz, Minimum HD: 10GB , Minimum Memory: 2GB

Table 11: Hardware and Software requirements for future experiment

According to information above, a virtual machine was setup at Green Lab with following settings:

- Operating System: Windows Server 2012 R2, 64bit
- CPU: Intel(R) Xeon(R) E5630 2.53GHz
- RAM: 4GB

On this server, IIS 8 was installed. All four open-source systems can be deployed on this version of IIS following their deployment tutorials that are reachable through the project website. Links to these websites were given in Chapter 4.

5.2 Experimentation Design

As given in Chapter 7, it is a recommended future work to conduct an experiment to analyze performance and energy efficiency of applications with and without detected anti-patterns. During the time for this thesis project, next to setting up an environment at the Green Lab of Vrije Universiteit van Amsterdam I also made the first steps towards design for such experiment. In this chapter, all information is given about what has been done, so that the work can be completed as future work, or future researchers can be inspired by this step towards realizing the experiment.

According to the suggested experimentation process by Wohlin00 et al., there are five steps that need to be followed by a successful experiment [Wohlin00]. These are: scoping, planning, operation, analysis & interpretation, and results. To make sure that this thesis project delivers valid and successful experimentation, the suggested process was followed.

Throughout steps of an experiment, following terms are suggested to be used [Wohlin00]. In this section you can find the meaning for these terms and their definitions within the context of the intended future experiment.

- Variables: There are two kinds of variables in an experiment, independent and dependent variables. Those variables that we want to study to see the effect of the changes in the independent variables are called dependent variables. All variables in a process that are manipulated and controlled are called independent variables [Wohlin00].

Therefore, for this experiment the dependent variables are **performance** and **energy consumption**, while the independent variables are **refactoring detected anti-patterns**.

- Treatments: An experiment studies the effect of changing one or more independent variables. Those variables are called factors. A treatment is one particular value of a factor [Wohlin00]. Treatments for this experiment are **the code piece that was detected as an anti-pattern** and

the resulting code piece after refactoring.

- Objects: The treatments are being applied to the combination of objects and subjects. An object can, for example, be a document that shall be reviewed with different inspection techniques [Wohlin00].

Objects for this future experiment can be the four Open-Source projects presented in Chapter 4 or other case studies of the researcher's choice. Criteria that was followed while choosing the open source projects in Chapter 4 are given below, so this can stand as inspiration for the future researchers:

1. Using a relational database to store data
 2. Using one of the three most common rich ORM tools for C# applications, which are also the focus for this thesis project (Entity Framework, LINQ to SQL, NHibernate)
 3. Projects are actively used in production.
- Subjects: The people that apply the treatment are called subjects.
For this thesis project the subject (or participant) is myself, the student performing the refactoring of detected anti-patterns and conducting measurement data. The developed anti-pattern detection tool can play a role as being the subject in the future by providing automatic refactoring after detection of the anti-patterns.

5.2.1 Scope of the Experiment

The purpose of the scoping is to define the goals of an experiment, which allow us to ensure that the intention with the experiment can be fulfilled through the experiment. [Wohlin00] suggests to use Goal Question Metric template, which was originally presented by [Basili88], to define the goals of an experiment.

The scope for the future experiment can be defined using this template as follows:

Analyze **code pieces, programming techniques**
for the purpose of **Evaluation of impact**
in terms of **Performance and Energy Efficiency**

from the point of view of **Application Developer, Architect**

in the context of **the master student detecting, refactoring code pieces on Open-Source C# applications and measuring the impact of the refactoring**

5.2.2 Experiment Planning

Given the list of anti-patterns from Chapter 3 of this thesis, by measuring performance and energy consumption impact of refactoring these anti-patterns, the main questions that are answered are:

- Are given ways of ORM usage actually performance anti-patterns?
- Are given ways of ORM usage actually energy-efficiency anti-patterns?

Metrics that can be used while answering these questions are Response Time for performance and Energy Consumption for energy-efficiency.

Given the treatments defined above for this future experiment, following hypothesis can be made for performance and energy-efficiency. If E = Energy-efficiency, P = Performance and “with” meaning with the code piece that was detected as an anti-pattern, “without” meaning the resulting code piece after refactoring,

- H0 (null hypothesis):
 - $E_{\text{with}} \leq E_{\text{without}}$
 - $P_{\text{with}} \leq P_{\text{without}}$
- H1 (alternative hypothesis):
 - $E_{\text{with}} > E_{\text{without}}$
 - $P_{\text{with}} > P_{\text{without}}$

After refactoring anti-patterns, the resulting improvements in response time and energy consumption can be different for different code execution paths within one project, because there are many factors that has an impact on this improvement. For example if in one code execution path an excessive data anti-pattern occurs that retrieves data for an entity that has extensive amount of relationships with other entities, the impact of fixing this anti-pattern for this call path would be greater than another

code execution path where an excessive data anti-pattern occurs on an entity that has few other entities related to it, meaning fewer data being retrieved from the database with the eager fetch. Given these facts, it is not possible to generalize any measurement for performance impact of fixing an anti-pattern as response time. However, it would be useful for developers to know fixing which anti-pattern would be more beneficial. Below example factors are listed for two anti-patterns, which would have an impact on response time improvement.

Excessive Data: Amount of entities related to the eagerly fetched entity, amount of and types of columns read for loading related entities

One-by-one Processing: Times with which the database call is made inside a loop (how many times the loop executes)

There are multiple ways each anti-pattern can be fixed. Factors that have an impact on response time improvement, for which examples are given above, should be taken into account. The chosen fix for each anti-pattern should be carefully chosen and explained.

Normality, t-test, Mann Whitney test and 5% significance ratio are suggested to be used while statistically analyzing the conducted measurements.

Chapter 6 – Analysis and Conclusions

6.1 Analysis and Interpretation of Results

The general aim of this study is to create a step forward towards extension of [Chen14]. In this original study, Chen et al. developed a framework for Java applications that automatically detects two ORM performance anti-patterns (Excessive Data and One-by-One Processing anti-patterns). With the use of the automatic detection tool they built, they were able to detect anti-patterns on three software systems. After doing statistically rigorous evaluation of performance impact of refactoring detected anti-patterns, they suggest significant performance improvement (up to %98 in response time) can be achieved in response time.

The original study focuses only on two ORM performance anti-patterns and conducts results on Java applications. To be able to draw more general conclusions on impact of refactoring ORM performance anti-patterns, it is necessary to look at other ORM performance anti-patterns. Several studies exist that focus on listing performance anti-patterns in general [Smitt00], [Smitt03], [Roussey09], [Tribuani14], [Tribuani15], detecting anti-patterns in general [Moha10] or detecting performance anti-patterns [Cortellessa10], [Parsons04]. However, none of them focuses specifically on providing a list of performance anti-patterns caused by ORM usage.

In this study the first attempt was made in this area to provide a list of ORM performance anti-patterns, according to considerations of developers in the field. The list, while not complete, contains all items that could be found within the given time limitations of this study.

After gathering the list of ORM performance anti-patterns, an ORM tool agnostic framework to allow automatic detection of these anti-patterns on C# applications was designed. As explained in Chapter 4, one of the findings of this study is that it is a very complex task to develop such a framework for C#. This is due to complexity of scenarios in which the listed ORM performance anti-patterns can exist and lack of useful frameworks with supporting documentation

Just like non-existence of studies providing a list of ORM performance anti-patterns, also none of the related studies make an attempt to generalize ORM performance anti-pattern detection in an ORM tool

agnostic framework.

In this study, an attempt was made to create an abstract object model to represent necessary information about a C# application, which can be used to detect ORM performance anti-patterns on this application. This abstract object model was validated against all three most common ORM tools for C# language and it was found that this model is applicable to usage of each of these ORM tools. This means that an ORM tool agnostic framework can be developed for C# applications for ORM performance anti-pattern detection purposes.

As main contribution of this study, the developed framework most importantly offers abstraction that allows detection of anti-patterns agnostic to ORM tool, and extraction of interesting data from C# applications. The framework stands as one of the few example projects for using Roslyn framework to interpret C# code. The contributions of this thesis provide a foundation for researchers who are willing to work further in this area. The in progress framework is available as a [public repository](#). When it is ready for distribution, it will be published with an open-source license.

6.2 Answers to Research Questions

RQ1 was “Which usages of ORM tools are considered performance anti-patterns in the literature?” This research question was answered with a list containing seven ORM performance anti-patterns. Except for two of the anti-patterns that were used in [Chen14], there are no studies that provide empirical data to support the expected decrease in performance due to implementation of the listed ways of using ORM tools. Therefore, validation of whether the listed ways of ORM usage actually are performance anti-patterns remains a question to be answered by future experiments.

RQ2 was “How can an ORM tool agnostic framework for automatically detecting ORM performance anti-patterns be built for C# applications?”

The designed abstract object model suggests that development of an ORM tool agnostic framework is possible. However, in this study only three of the most common rich ORM tools for C# applications were included while validating this abstraction. Analysis of whether same object model would be applicable to other rich ORM tools and whether it would be applicable to light ORM tools is necessary to be able to draw conclusions on possibilities of developing an ORM tool agnostic framework.

The developed ORM performance anti-pattern detection tool shows that most of the data to detect listed anti-patterns on C# applications can be extracted using syntactic and semantic analysis of source code. Though, technical challenges such as to extract SQL queries that are generated by the ORM tool in run-time, or to serialize extracted data to improve performance of the detection tool pose threats to validity while answering RQ2. While it consumes time due to lack of documentation, Roslyn framework exists to help perform such analysis. Next to these, much of the functionality that are required to complete the framework are implemented such as generation of code execution paths and detection rule execution.

Chapter 7 - Recommendations for Future Work

Listed ways of ORM usage in Chapter 3 are considered as performance anti-patterns by developers in the industry. However, there is not enough research which shows performance impact of refactoring suggested anti-patterns. Therefore, more empirical studies must be done to draw clearer conclusions in this regard.

Next to performance, energy consumption impact of refactoring these anti-patterns is also a point of interest to other researchers. Lowering energy consumption of software systems, increasing sustainability is a research area. Interest to extend this study by also measuring energy consumption impact is interesting for researchers in this area. As part of the future work, firstly more literature study must be done to understand work done on development of energy efficient software. Then experiments must be done to analyze energy consumption impact of refactoring suggested anti-patterns. These future experiments will also produce empirical data to strengthen conclusions on the relation between performance and energy consumption.

A laboratory called Green Lab is made available by Prof. Dr. Patricia Lago and Dr. Giuseppe Procaccianti from Vrije Universiteit Amsterdam, to help conduct an experiment to measure both performance and energy consumption impact of refactoring anti-patterns listed in this study.

Approximately 160 hours within this project were spent on setting up a virtual environment on servers in Green Lab and to plan an experiment for measuring performance and energy consumption impact of refactoring two of the listed anti-patterns (Excessive Data and One-by-One Processing). Due to already mentioned challenges and time limitation, this experiment has not yet taken place. Though, it is highly recommended to be done as future work. What has been already setup and planned is explained in Chapter 5, which can be used as a first step for future researchers.

Before conducting the mentioned experiment, anti-pattern detection on the case studies must be completed. One approach is to manually detect anti-patterns, or use the manual detection analysis results I presented for the chosen case studies, and focus on measurement of performance and energy consumption impact of refactoring detected anti-patterns. Another approach is to invest further in completing the automatic detection framework that was presented in Chapter 4. Since

automatic detection would be repeatable, quick and would result in more accurate analysis, this approach is highly recommended.

While developing the automatic detection tool further, several technical challenges must be overcome. Firstly, to be able to detect Excessive Data, One-by-One Processing and Inefficient Lazy Loading anti-patterns, it is necessary to know which entities a Database Accessing Select Call will return. There are several ways a call can be configured to eagerly load related entities. This can be on the level of Data Context Declaration, Database Entity Declaration or Database Accessing Select Call Statement and the configuration required differs between ORM tools. To detect this configuration means checking all possible levels by syntactical analysis. If we look at what we are trying to do, actually we just want to know which related entities will be loaded together with the main selected entity when a Database Accessing Select Call is done. If the related entity is used in the rest of the code execution path or not, this will determine the existence of one of these three anti-patterns. Another approach to know this information for a given Database Accessing Select Call Statement is to know which SQL query will be generated and executed by the ORM tool. When LINQ is used as query language, it is possible in the run-time to use ToString () method on the query and see the generated SQL query. This mechanism uses the implementation provided by ORM tool's LINQ provider. A similar analysis was done by [Albahari15] for development of LINQPad, a tool that helps developers quickly see execution result of LINQ queries. LINQPad constructs an in memory Data Context Declaration and Database Entity Declarations, then let's developers use these to quickly type a LINQ query and gathers results of this query from an RDBMS that the developer pointed at. While LINQPad configures lazy loading as default and has no concern to extract this configuration, in our tool we have to analyze the choice of the developers on this regard. Generating SQL query which will be executed in run-time according to configuration is a challenge to be tackled.

For detection of other anti-patterns, data extraction must be extended to detect all types of objects which are defined in the abstract object model. For this, the existing part of the framework can be used as example and syntactic and semantic analysis data will be sufficient to gather these objects. One of the included ORM tools is NHibernate, but due to time limitations it was not possible yet to extract data on applications using this ORM tool. Classes must be implemented based on the

abstraction done for data extraction to add capabilities to the detection framework for extraction of objects in the abstract object model for NHibernate. It is important to pay attention to NHibernate version used in case studies while implementing these classes, since the configuration can be different between versions.

Appendix A - Source Code of Anti-Pattern Detection Framework

In this appendix some of the important source code of ORM performance anti-patterns detection tool are listed. The entire project can be found on this public repository:

<https://github.com/TubaKayaDev/ORMPerformanceAntiPatternDetector>.

A.1 Object Model

In Detector.Models project, you can find interfaces and classes representing the model.

For each ORM tool that are targeted, a class represents this tool and is used as the generic type while defining model classes.

```
public sealed class EntityFramework : ORMToolType
{
}
```

Code Sample 13: Class representing Entity Framework tool type in detection framework

Each model from abstract object model that was presented in Chapter 4, derive from following interface and abstract class:

```
public interface Model
{
    CompilationInfo CompilationInfo { get; }
}

public abstract class ModelBase : Model
{
    public CompilationInfo CompilationInfo { get; private set; }

    public ModelBase(CompilationInfo compilationInfo)
    {
        CompilationInfo = compilationInfo;
    }
}
```

```
}  
  
}
```

Code Sample 14: Base Model implementation in detection framework

The models are split into logical folders. An example model class is `DataContextDeclaration`, which derives from `DeclarationBase` and resides in folder `ORM\DataContexts`. Code Sample 15 shows its implementation:

```
public abstract class DeclarationBase : ModelBase  
{  
    public string Name { get; private set; }  
  
    public DeclarationBase(string name, CompilationInfo compilationInfo)  
        : base(compilationInfo)  
    {  
        Name = name;  
    }  
}  
  
public class DataContextDeclaration<T> : DeclarationBase  
{  
    public DataContextDeclaration(string name, CompilationInfo compilationInfo)  
        : base(name, compilationInfo)  
    { }  
}
```

Code Sample 15: A concrete model class (`DataContextDeclaration`) in detection framework

A.2 Data Extraction

Currently there are abstract classes in `Detector.Extractors.Base` library, which implement common functionality for extraction of each model such as:

```

public abstract class Extractor<T> where T : ORMToolType
{
    public Context<T> Context { get; }

    public Extractor(Context<T> context)
    {
        Context = context;
    }
}

public abstract class DataContextDeclarationExtractor<T> : Extractor<T> where T :

    ORMToolType
{
    public HashSet<DataContextDeclaration<T>> DataContextDeclarations { get; protected set;
                                                                    }

    public DataContextDeclarationExtractor(Context<T> context)
        : base(context)
    {
        DataContextDeclarations = new HashSet<DataContextDeclaration<T>>();
    }

    public async Task FindDataContextDeclarationsAsync(Solution solution,
                                                         IProgress<ExtractionProgress> progress)
    {
        await ExtractDataContextDeclarationsAsync(solution, progress);

        Context.DataContextDeclarations = DataContextDeclarations;
    }
}

```

```
protected abstract Task ExtractDataContextDeclarationsAsync(Solution solution,
                                                           IProgress<ExtractionProgress> progress);
}
```

Code Sample 16: An abstract base Extractor implementation for Data Context Declarations

Then corresponding concrete class in one of the libraries for extraction on specific ORM tool, such as Detector.Extractors.EF602 (extracts data for applications which use Entity Framework version 6.02):

```
public class DataContextDeclarationExtractor :
    DataContextDeclarationExtractor<EntityFramework>
{
    public DataContextDeclarationExtractor(Context<EntityFramework> context)
        : base(context)
    { }

    protected override async Task ExtractDataContextDeclarationsAsync(Solution solution,
                                                                       IProgress<ExtractionProgress> progress)
    {
        string extractionNote = "Extracting Data Context Declarations by finding classes of
                                type DbContext";
        progress.Report(new ExtractionProgress(extractionNote));

        Dictionary<ClassDeclarationSyntax,SemanticModel> classes = await
                                                                    solution.GetClassesOfType<DbContext>();

        foreach (var item in classes.Keys)
        {
            DataContextDeclarations.Add(new
                                        DataContextDeclaration<EntityFramework>(item.Identifier.ToString(),
```



```

        item.GetCompilationInfo(classes[item]));
    }
}

```

Code Sample 17: A concrete Extractor implementation

A refactoring for this part of the system is in progress. Implementing extraction algorithms on Roslyn solutions can be made reusable by using Strategy pattern [Gamma95]. For this purpose, refactoring has been started but not yet completed. Nevertheless, here is an example how the code is intended to be changed in the future to allow more re-usability. Following class is an example Strategy to extract `ClassDeclarationSyntax` nodes from Roslyn solution which are derived from a provided class or interface name:

```

internal class ClassDeclarationSyntaxExtractionBasedOnClassDerivation<T> :
    ExtractionStrategy<ClassExtractionReturnType> where T : ORMToolType
{
    private SolutionParameter _solutionParameter;
    private StringParameter _derivedFromTypeNameParameter;

    internal ClassDeclarationSyntaxExtractionBasedOnClassDerivation(params Parameter[]
                                                                    parameters)
        : base(parameters)
    {}

    internal override void SetParameters(params Parameter[] parameters)
    {
        _solutionParameter = parameters.Where(p => p is SolutionParameter) as
                                                                    SolutionParameter;
        _derivedFromTypeNameParameter = parameters.Where(p => p is StringParameter) as
                                                                    StringParameter;
    }
}

```

```

internal override async Task<HashSet<ClassExtractionReturnType>> Execute()
{
    var dataContextDeclarations = new HashSet<DataContextDeclaration<T>>();
    Dictionary<ClassDeclarationSyntax, SemanticModel> classes = await
        _solutionParameter.Value.GetClassesOfType(_derivedFromTypeNameParameter.Value);

    var result = new HashSet<ClassExtractionReturnType>();
    foreach (var item in classes.Keys)
    {
        result.Add(new ClassExtractionReturnType(classes[item], item));
    }
    return result;
}
}

```

Code Sample 18: An implementation of extraction strategy

Then a general ExtractorBuilder class was implemented (in progress). This class can be used to build the ORM tool type specific extractor classes. This class is shown in Code Sample 19 and 20.

```

public class ExtractorBuilder<T> where T : ORMToolType
{
    private SolutionParameter _solutionParameter;

    private ExtractionStrategy<ClassExtractionReturnType>
        _dataContextDeclarationExtractionStrategy;
    private ExtractionStrategy<ClassExtractionReturnType>
        _databaseEntityDeclarationExtractionStrategy;
    private ExtractionStrategy _databaseAccessingMethodCallStatementExtractionStrategy;

    private HashSet<DataContextDeclaration<T>> _dataContextDeclarations;

```

```

private HashSet<DatabaseEntityDeclaration<T>> _databaseEntityDeclarations;

public ExtractorBuilder(Solution solution)
{
    _solutionParameter = new SolutionParameter("solution", solution);
}

public ExtractorBuilder<T> WithDataContextDeclarationsDerivedFromBaseClass(string
    baseClassName)
{
    var baseClassNameParameter = new StringParameter("baseClassName", baseClassName);

    _dataContextDeclarationExtractionStrategy = new
        ClassDeclarationSyntaxExtractionBasedOnClassDerivation<T>(_solutionParameter,
                                                                    baseClassNameParameter);

    return this;
}

```

Code Sample 19: ExtractorBuilder class to be used with extraction strategies

```

public ExtractorBuilder<T> WithDataContextDeclarationsSignedWithAttribute(string
    attributeName)
{
    ...
}

public ExtractorBuilder<T> WithDbEntitiesDerivedFromBaseClass(string baseClassName)
{
    ...
}

```

```

}

public ExtractorBuilder<T>
    WithDbEntitiesAsGenericTypeOnIQueryablesInDataContextClasses()
{
    ...
}

public async Task Extract()
{
    var resultDataContextExtractions = await
        _dataContextDeclarationExtractionStrategy.Execute();
    _dataContextDeclarations = new HashSet<DataContextDeclaration<T>>();
    foreach (var item in resultDataContextExtractions)
    {
        var dataContextDeclaration = new
            DataContextDeclaration<T>(item.ClassDeclarationSyntax.Identifier.ToString(),

item.ClassDeclarationSyntax.GetCompilationInfo(item.SemanticModel));
        _dataContextDeclarations.Add(dataContextDeclaration);
    }
    var resultDatabaseEntities = await
        _databaseEntityDeclarationExtractionStrategy.Execute();
    _databaseEntityDeclarations = new HashSet<DatabaseEntityDeclaration<T>>();
    foreach (var item in resultDatabaseEntities)
    {
        var databaseEntityDeclaration = new
            DatabaseEntityDeclaration<T>(item.ClassDeclarationSyntax.Identifier.ToString(),
            item.ClassDeclarationSyntax.GetCompilationInfo(item.SemanticModel));
        _databaseEntityDeclarations.Add(databaseEntityDeclaration);
    }
}

```

```

        await _databaseAccessingMethodCallStatementExtractionStrategy.Execute(parameters);
    }
}

```

Code Sample 20: ExtractorBuilder continue

These classes implementing Strategy Pattern are placed under project RoslynSyntaxNodeExtractors and are recommended to be further implemented in the future.

Next to Strategy Pattern, the way progress information reporting implemented can also be refactored to use Observer Pattern. At the moment each extraction method expects an `IProgress<ExtractionProgress>` object and reports on this object while processing. Instead, implementing a collection of observers for extractors would allow more flexibility in the future. In this case, for different observers such as an error logger, console logger or WPF reporter, a new observer class can be implemented and adding an object of type in this class to the collection would result in reporting to that object as well. Therefore, extractors can publish information about their progress when steps are executed and the observers can report as implemented. This implementation is recommended to be done as future work.

Unit tests are written to validate the correctness of implemented functionality. Under Tests folder in the solution, a test project can be found for each project. Also ProjectsUnderTest folder contains some example projects using LINQ to SQL and Entity Framework. Since extraction methods require a Roslyn solution object, for some tests these example projects were used while creating Roslyn solution for a test.

Example test for an extractor is given in Code Sample 21:

```

[TestClass]
public class DataContextDeclarationExtractorTests
{
    [TestMethod]
    [TestCategory("IntegrationTest")]
    public async Task DetectsDbContextClasses_When_EF60_NWProjectIsCompiled()

```

```

{
    //Arrange
    Solution EF60_NWSolution = await new RoslynSolutionGenerator().GetSolutionAsync(
        @"..\..\..\ProjectsUnderTest\EF60_NW\EF60_NW.sln");

    Context<EntityFramework> context = new ContextStub<EntityFramework>();

    var target = new DataContextDeclarationExtractor(context);

    var progressIndicator = new ProgressStub();

    //Act
    await target.FindDataContextDeclarationsAsync(EF60_NWSolution, progressIndicator);

    //Assert
    var item = target.DataContextDeclarations.First();

    Assert.IsTrue(target.DataContextDeclarations.Count == 1);
    Assert.IsTrue(item.Name == "NWDbContext");
    Assert.IsTrue(target.DataContextDeclarations.Count == 1);
    Assert.IsTrue(context.DataContextDeclarations == target.DataContextDeclarations);
}
}

```

Code Sample 21: Unit test method for Data Context Declaration Extractor

A.3 Detection Rules

Detector.Main project contains detection rules. Each detection rule derives from the base class:

```

public abstract class DetectionRule
{
    protected CodeExecutionPath CodeExecutionPath { get; private set; }

    public bool AppliesToModelTree(CodeExecutionPath codeExecutionPath)
    {
        this.CodeExecutionPath = codeExecutionPath;
        return GetRuleFunction().Invoke();
    }

    public List<AntiPatternBase> DetectedAntiPatterns { get; protected set; }

    protected abstract Func<bool> GetRuleFunction();

    public DetectionRule()
    {
        this.DetectedAntiPatterns = new List<AntiPatternBase>();
    }
}

```

Code Sample 22: Base Detection Rule class

For each concrete detection rule a new class is implemented. An example is

OneByOneProcessingDetectionRule:

```

public class OneByOneProcessingDetectionRule<T> : DetectionRule where T : ORMToolType
{
    protected override Func<bool> GetRuleFunction()
    {
        return PathHasLazyLoadingDbAccessingMethodCall
            AndRelatedEntitiesAreCalledOnReturnedObject;
    }
}

```

```

    }

    public bool PathHasLazyLoadingDbAccessingMethodCall
        AndRelatedEntitiesAreCalledOnReturnedObject()
    {
        IEnumerable<DatabaseAccessingMethodCallStatement<T>> databaseAccessingMethodCalls
=
        CodeExecutionPath.Models.OfType<DatabaseAccessingMethodCallStatement<T>>();

        IEnumerable<DatabaseEntityVariableRelatedEntityCallStatement<T>>
            databaseEntityVariableRelatedEntityCalls =
        CodeExecutionPath.Models.OfType<DatabaseEntityVariableRelatedEntityCallStatement<T>>()
;

        foreach (var dbAccessingMethodCall in databaseAccessingMethodCalls)
        {
            if (!dbAccessingMethodCall.DoesEagerLoad)
            {
                if (databaseEntityVariableRelatedEntityCalls.Any(x =>
                    x.CalledDatabaseEntityVariable == dbAccessingMethodCall.AssignedVariable))
                {
                    this.DetectedAntiPatterns.Add(new
                        OneByOneProcessingAntiPattern(CodeExecutionPath, dbAccessingMethodCall));
                    return true;
                }
            }
        }
        return false;
    }
}

```

Code Sample 23: An example concrete Detection Rule class (One-by-One Processing Detection Rule)

An example test for this rule is in Detector.Main.Tests project and given below. Please note that the CodeExecutionPathGenerator class implements Builder Pattern and resides in DetectionRules\Helpers folder under same test project.

```
[TestClass]
public class OneByOneProcessingDetectionRuleTests
{
    OneByOneProcessingDetectionRule<FakeORMToolType> target;

    [TestInitialize]
    public void Initialize()
    {
        target = new OneByOneProcessingDetectionRule<FakeORMToolType>();
    }

    [TestMethod]
    public void DetectsOneByOneProcessingAntiPattern_When_
        ThereIsOneQueryInTheTreeThatDoesLazyLoading
        AndRelatedEntityIsUsedInALoop()
    {
        //Arrange
        var codeExecutionPath = new CodeExecutionPathGenerator()
            .WithLazyLoadingDatabaseAccessingMethodCall()
            .WithDatabaseEntityVariableAssignedByDatabaseAccessingMethodCall()
            .WithCallToRelatedEntityOnDatabaseEntityVariable
                AssignedByDatabaseAccessingMethodCall()
            .Build();

        ///Act
        bool result = target.AppliesToModelTree(codeExecutionPath);
    }
}
```

```

        ////Assert
        Assert.IsTrue(result);
    }
}

```

Code Sample 24: Unit test for One-by-One Detection Rule implementation

A.4 Illustration of Extractors

In Detector.Main project, another important class is Extraction Manager. This class receives concrete extractor objects in its constructor and illustrates the steps of executing extraction through the objects. Currently Newtonsoft library is used to serialize data into JSON files. Extraction Manager also calls methods to trigger serialization and storing of the serialized data into JSON files. A concrete object of class NewtonsoftSerializer was also sent to Extraction Manager to be used for this purpose.

```

public class ExtractionManager<T> : IExtractionManager where T : ORMToolType
{
    private DataContextDeclarationExtractor<T> _dataContextDeclarationExtractor;
    private DatabaseEntityDeclarationExtractor<T> _databaseEntityDeclarationExtractor;
    private DatabaseAccessingMethodCallExtractor<T> _databaseAccessingMethodCallExtractor;
    private CodeExecutionPathExtractor<T> _codeExecutionPathExtractor;
    IProgress<ExtractionProgress> _progressIndicator;
    private ISerializer<T> _serializer;
    public ExtractionManager(
        DataContextDeclarationExtractor<T> dataContextDeclarationExtractor
        , DatabaseEntityDeclarationExtractor<T> databaseEntityDeclarationExtractor
        , DatabaseAccessingMethodCallExtractor<T> databaseAccessingMethodCallExtractor
        , CodeExecutionPathExtractor<T> codeExecutionPathExtractor
        , IProgress<ExtractionProgress> progressIndicator
        , ISerializer<T> serializer)
    {

```

```

        _dataContextDeclarationExtractor = dataContextDeclarationExtractor;
        _databaseEntityDeclarationExtractor = databaseEntityDeclarationExtractor;
        _databaseAccessingMethodCallExtractor = databaseAccessingMethodCallExtractor;
        _codeExecutionPathExtractor = codeExecutionPathExtractor;
        _progressIndicator = progressIndicator;
        _serializer = serializer;
    }

    public async Task ExtractAllAsync(string solutionUnderTest, string folderPath)
    {
        ...

        await ExtractDataContexts(solutionUnderTest, solution);
        await ExtractDatabaseEntities(solutionUnderTest, solution);
        await ExtractDatabaseAccessingMethodCalls(solutionUnderTest, solution);
        await GenerateCodeExecutionPaths(solutionUnderTest, solution);
    }

    ...

    ...
}

```

Code Sample 25: Class illustrating execution of all extractions in order

A.5 Console Application

Detector.ConsoleApp instantiates ExtractionManager instance after resolving the dependencies and invokes ExtractAllAsync method. Following is the class that resolves dependencies:

```

public class DependencyResolver
{
    public IExtractionManager GetExtractionManager()
    {
        var context = new ConcreteContext<EntityFramework>();
    }
}

```

```

var dataContextDecExt = new DataContextDeclarationExtractor(context);
var dbEntityExt = new
    DatabaseEntityDeclarationExtractorUsingDbContextProperties(context);
var dbAccessingMethodCallExt = new DatabaseAccessingMethodCallExtractor(context);
var codeExecutionPathExt = new CodeExecutionPathExtractor(context);
var progressIndicator = new Progress<ExtractionProgress>((e) =>
    ProgressChanged(e));
var serializer = new NewtonsoftSerializer<EntityFramework>();

IExtractionManager extractionManager = new
    ExtractionManager<EntityFramework>(dataContextDecExt, dbEntityExt,
        dbAccessingMethodCallExt, codeExecutionPathExt, progressIndicator,
serializer);
    return extractionManager;
}
private void ProgressChanged(ExtractionProgress extractionProgress)
{
    if (!string.IsNullOrEmpty(extractionProgress.ExtractionType))
    {
        Console.WriteLine();
        Console.WriteLine(extractionProgress.ExtractionType);
    }
    else
    {
        Console.Write(" \r{0}% done ", extractionProgress.PercentageOfWorkDone);
    }
}
}

```

Code Sample 26: Resolving dependencies in Console Application

Then in program class, the resolved ExtractionManager was called:

```
class Program
{
    static void Main(string[] args)
    {
        Task t = MainAsync(args);
        t.Wait();
        Console.WriteLine("Press enter to exit...");
        Console.Read();
    }
    static async Task MainAsync(string[] args)
    {
        Console.WriteLine("Tool started at " + DateTime.Now.ToLongTimeString());
        string folderPath = @"D:\School\Thesis Bug prediction with
                               antipatterns\Projects\vc-community\PLATFORM";
        string solutionUnderTest = "VirtoCommerce.WebPlatform";

        Console.WriteLine("Extracting for solution: " + folderPath + @"\" +
                               solutionUnderTest);

        var extractionManager = new DependencyResolver().GetExtractionManager();

        await extractionManager.ExtractAllAsync(solutionUnderTest, folderPath);
        Console.WriteLine("Tool stopped at " + DateTime.Now.ToLongTimeString());
    }
}
```

Code Sample 27: Main class in Console Application

Appendix B - Extracted Data Examples

In this appendix, the extracted data in JSON format for Data Context Declarations, Database Entity Declarations and a part of the Database Accessing Method Call Statements are listed for VirtoCommerce.WebPlatform solution. Please find the complete list in CaseStudyResults folder on the public repository:

<https://github.com/TubaKayaDev/ORMPerformanceAntiPatternDetector/tree/master/CaseStudyResults/VirtoCommerce.WebPlatform>.

B.1 Data Context Declarations

Structure for Data Context Declaration data contains

- Name of the class that is declared as a data context and
- The Syntax Node Location being the physical location of the file on which this class is written.

Following are two items from the results extracted for VirtoCommerce.WebPlatform solution:

```
[{
  "Name": "EFRepositoryBase",
  "CompilationInfo": {
    "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Infrastructure\\EFRepositoryBase.cs[620
..11395))"
  }
},
{
  "Name": "PlatformRepository",
  "CompilationInfo": {
```

```

    "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Repositories\\PlatformRepository.cs[430
..6375))"
  }
},
...]
```

Extracted Data Sample 1: Data Context Declaration data

B.2 Database Entity Declarations

Structure for Database Entity Declaration data contains

- Name of the class that is declared as a data context and
- The Syntax Node Location being the physical location of the file on which this class is written.

Following are two items from the results extracted for VirtoCommerce.WebPlatform solution:

```

[
  {
    "Name": "SettingEntity",
    "CompilationInfo": {
      "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Model\\SettingEntity.cs[217..1422))"
    }
  },
  {
    "Name": "AccountEntity",
    "CompilationInfo": {
```

```

    "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Model\\AccountEntity.cs[136..808))"
  }
},
...]
```

Extracted Data Sample 2: Database Entity Declaration data

B.3 Database Accessing Method Call Statements

Structure for Database Accessing Method Call Statements data contains more elements than what the current version of the tool can extract. The data that is currently possible to extract are:

- Entity Declarations Used in Query are list of Database Entity Declarations, which are declarations of types of entities that are expected to be returned by the Database Accessing Method Call Statement
- Query Text In C# being the line on which the Database Accessing Method Call Statement was found.
- Parent Method Name is the name of the method which contains the detected Database Accessing Method Call Statement within its body blocks. This data is used for generation of Code Execution Paths.
- The Syntax Node Location being the physical location of the file on which this class is written.

Following are two items from the results extracted for VirtoCommerce.WebPlatform solution:

```

[
{
  "ExecutedQuery": null,
  "EntityDeclarationsUsedInQuery": [
    {
```



```

    "Name": "OperationLogEntity",
    "CompilationInfo": {
      "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Model\\OperationLogEntity.cs[261..532)
)"
    }
  },
  "DatabaseQueryVariable": null,
  "QueryTextInCSharp": "\\t\\t\\tretVal.InjectFrom(entity)",
  "DataContext": null,
  "LoadedEntityDeclarations": [],
  "DoesEagerLoad": false,
  "AssignedVariable": null,
  "ParentMethodName": "ToCoreModel",
  "CompilationInfo": {
    "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\ChangeLog\\ChangeLogConverter.cs[513
..538))"
  }
}
... ]

```

Extracted Data Sample 3: Database Accessing Method Call Statement data

B.4 Code Execution Paths

Structure for Code Execution Path data contains Models listed in the Code Execution Path. Currently a

MethodCall model is created for each call found in the solution that accesses a method (here and after called Parent Method) which contains a Database Accessing Method Call Statement. This is done by analyzing syntactic and semantic information for invocations of methods which call either the Parent Method's declaration on the class or on the interface.

A MethodCall object has following data:

- The Syntax Node Location being the physical location of the file on which this class is written.
- Method Containing Call being the caller
- CalledMethodName being the Parent Method.

Following are two items from the results extracted for VirtoCommerce.WebPlatform solution:

```
[ { "Models": [
  {
    "CompilationInfo": {
      "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
antipatterns\\Projects\\vc-
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Infrastructure\\ServiceBase.cs[2075..212
2))"
    },
    "MethodContainingCall": "SaveObjectSettings",
    "CalledMethodName": "SaveSettings"
  }
],
{
  "Models": [
    {
      "CompilationInfo": {
        "SyntaxNodeLocation": "SourceFile(D:\\School\\Thesis Bug prediction with
```

```
antipatterns\\Projects\\vc-  
community\\PLATFORM\\VirtoCommerce.Platform.Data\\Infrastructure\\ServiceBase.cs[670..736)  
)"  
    },  
    "MethodContainingCall": "LoadObjectSettings",  
    "CalledMethodName": "GetObjectSettings"  
  }  
]  
}]
```

Extracted Data Sample 4: Code Execution Path data containing MethodCall model

Bibliography

- [Chen14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, *“Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping”*, Proceedings of the 36th International Conference on Software Engineering ICSE, 2014
- [Yvonne14] Yvonne N. Bui, *“How to Write a Master's Thesis 2. Edition”*, Sage Publications Inc. 2014
- [Wohlin00] Claes Wohlin, Per Runeson, Martin Höst, Dr. Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén, *“Experimentation in Software Engineering”*, Sweden 2000
- [Leavitt00] Neal Leavitt, *“Whatever happened to object-oriented databases?”*, Computer, 33(8):16–19, August 2000
- [Meyer97] Bertrand Meyer, *“Object-Oriented Software Construction Second Edition”*, Interactive Software Engineering Inc., 1997
- [Smith87] Karen E. Smith, Stanley B. Zdonik, *“Intermedia: A Case Study of the Differences between Relational and Object-Oriented Database Systems”*, OOPSLA'87 Proceedings, October 4-8, 1987
- [Dearle10] Alan Dearle, Roberto V. Zicari, *“Objects and Databases”*, Third International Conference, ICOODB 2010 Frankfurt/Main, Germany, September 28-30, 2010
- [Kulkarni07] Dinesh Kulkarni, Luca Bolognese, Matt Warren, Anders Hejlsberg, Kit George, *“LINQ to SQL, .NET Language-Integrated Query for Relational Data”*, Microsoft Corporation, February 2007
- [TIOBE15] *“TIOBE Index for August 2015”*,
<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, 11 August 2015
- [Cvetković10] Stevica Cvetković, Dragan Janković, *“A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment”*, ICOODB 2010, LNCS 6348, pp. 147–158, 2010. © Springer-Verlag Berlin Heidelberg 2010
- [Mitchell15] Scott Mitchell, *“An Extensive Examination of Data Structures”*,
[https://msdn.microsoft.com/en-us/library/aa289149\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa289149(v=vs.71).aspx), 14 July 2015
- [Tip00] Frank Tip, Jens Palsberg, *“Scalable Propagation-Based Call Graph Construction Algorithms”*, in

proceedings of OOPSLA'00, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 281–293, 2000

[Meuller13] John Paul Meuller, *“Step by Step Microsoft ADO.Net Entity Framework”*, Microsoft Press, August 2013

[Schenker11] Gabriel N. Schenker, *“NHibernate 3 Beginner's Guide”*, Packt Publishing, August 2011

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *“Design Patterns: Elements of Reusable Object-Oriented Software”*, Reading, MA, Addison-Wesley, 1995

[Galloway15] Jon Galloway, Scott Hanselman, *“Microsoft-supported open source projects”*, <http://www.asp.net/mvc/open-source>, 01-05-2015

[Mehta08] Vijay P. Mehta, *“Pro LINQ Object Relational Mapping with C# 2008”*, A-Press, 2008

[Bass13] Len Bass, Paul Clements, Rick Kazman, *“Software Architecture in Practice, Third Edition”*, Pearson Education, Inc. 2013

[Albahari15] Joseph Albahari, *“Why LINQ beats SQL”*, <https://www.linqpad.net/WhyLINQBeatsSQL.aspx>, 5 August 2015

[Basili88] Basili, V.R., Rombach, *“H.D.: The TAME project: towards improvement-oriented software Environments”*, IEEE Trans. Softw. Eng. **14**(6), 758–773, 1988

[Smith00] Connie U. Smith, Lloyd G. Williams, *“Software Performance Anti-Patterns”*, Proceedings 2nd International Workshop on Software and Performance, September 2000

[Tribuani15] Catia Trubiani, *“Introducing Software Performance Anti-patterns in Cloud Computing Environments: Does it Help or Hurt?”*, ICPE'15, Jan. 31–Feb. 4, Austin, Texas, USA, 2015

[Trubiani14] Catia Trubiani, Antinisca Di Marco, Vittorio Cortellessa, Nariman Mani, Dorina Petriu, *“Exploring Synergies between Bottleneck Analysis and Performance Anti-patterns”*, ICPE'14, Dublin, Ireland, March22–26, 2014

[Smith03] C. U. Smith, L.G. Williams, *“More new software performance anti-patterns: Even more ways to shoot yourself in the foot.”*, In Proceedings of the 2003 Computer Measurement Group Conference, CMG 2003, 2003

[Roussey09] Catherine Roussey, Oscar Corcho, Luis Manuel Vilches Blazquez, “A Catalogue of OWL Ontology AntiPatterns”, K-CAP’09, Redondo Beach, California, USA, September 1–4, 2009

[Moha10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur, “*DECOR: A Method for the Specification and Detection of Code and Design Smells*”, IEEE Transactions on Software Engineering, Vol.36, No.1, January/February 2010

[Cortellessa10] Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo, “*Digging into UML models to remove Performance Anti-patterns*”, QUOVADIS ’10, Cape Town, South Africa, May 3 2010

[Parsons04] Trevor Parsons, “*A framework for detecting, assessing and visualizing performance anti-patterns in component based systems*”, OOPSLA’04, Vancouver, British Columbia, Canada, Oct. 24–28, 2004