Reducing coupling to lower maintenance effort

Hans van Bakel

17 August 2012

One year Master Software Engineering

Thesis Supervisor : Jurgen Vinju

Company Supervisor: Chiel Labee

Host company: VisionWaves

Publication Status: Public

University of Amsterdam

Contents

Abstract

As the complexity of software systems increases their maintainability decreases. This is problematic since the majority of the total cost of a software system is related to maintenance. Many metrics have been proposed in order to measure the maintainability of a software system. However, there is a lack of quantitative results providing insight in the benefits of targeting specific metrics early in the development process.

Coupling is a concept specific to object-oriented languages that can be measured by various metrics. This thesis validates if lowering the coupling of an existing application and executing predefined maintenance scenarios on the original and altered system will ease maintenance. The level of increased ease of maintenance can be used to determine how much up-front design is justified.

This thesis focusses on the benefits when only coupling is lowered from the viewpoint of maintenance.

The results show no indication that lowering coupling is beneficial to the maintainability of a software system directly. Loosely coupled but highly cohesive modules are extracted. This isolation is beneficial to both testability as well as understandability, which influence maintainability indirectly.

Preface

The subject of this thesis is something that I have found intriguing for quite some time. Therefore the subject of my thesis was clear very quickly. However, to come up with a proper question that was well defined and fitted the requirements was harder. I want to thank Jurgen Vinju for all his help during my thesis, mainly for challenging the assumptions I had beforehand.

I want to thank Morgane Milienne-Petiot, Chiel Labee, Nick van Gils and Tijs van der Storm for their effort on verifying (early) versions of my thesis and providing valuable feedback and Craig Wilkinson for proof-reading an early version.

Chapter 1

Introduction

The maintenance part of the software life-cycle of a project is a major contributor to the overall cost of the project. This thesis will check whether reducing coupling lowers the maintenance effort needed to perform predefined maintenance tasks.
Previous research has shown that most of the cost of developing a software system is not related to the construction phase of the system but to the maintenance phase thereof [2]. This justifies adjusting the construction in order to improve the so-called 'Ease of maintenance'. Ease of maintenance is something that is hard to measure however there are metrics that can be used to get a rough sense of the maintainability of a system [5] [19]. Also, during maintenance activities, it is very important for developers to get a quick and accurate understanding of what the software system is supposed to do in order to perform the maintenance correctly [21].

1.1 Definitions

Some definitions apply to maintenance and ease of maintenance as these terms are widely used in various ways with different meanings. Coupling and cohesion are explained below.
Maintenance is defined by IEEE[1] as "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment". Based on this

[1]Defined in "IEEE Standard Glossary of Software Engineering Terminology"

definition, a system is more easy to maintain when it is easier to make the required adjustments. When maintenance is carried out, the software system is adjusted to accommodate for the new needs.
Ease of maintenance describes the effort that is needed to perform a specific maintenance task. Performing maintenance is about more than just adjusting the code needed to reflect the change, pinpointing the correct place to make the adjustments is sometimes more difficult than the actual modification. Ease of maintenance covers the entire process starting from a request to change the code to the result of a changed codebase that reflects the desired change. A system that is easier to maintain has a higher ease of maintenance.

1.2 Coupling

Coupling is a principle in programming that signals that a single element is dependent (or coupled) upon another element. Coupling can be defined as: the amount and level of dependency of a single element upon other elements. These elements can be classes or other (sub)systems, for the remainder of this chapter 'class' will be used to make it more concrete.
The functionality of a class can be broken when the dependent class is changed. This is undesirable as small changes can have far reaching effects that are unforeseen and unrelated to the change. To measure coupling, various metrics have been proposed[3]. These metrics calculate a value for every class in the system which indicates how dependent a class is upon other classes. A growing number of dependencies indicates an increasing likelihood for the functionality of the class to be broken by changes made to other classes.

1.2.1 Strength of coupling

Besides the number of dependencies, every dependency has a certain strength associated with it. The strength of a dependency indicates how interrelated these two classes are. A high value for strength means two classes use each others methods and/or types very frequently. As a result, lowering coupling between two classes that are strongly coupled is more complex. The strength of coupling is influenced strongly by the way two classes are coupled:

God class in this case there is only a single class. Multiple classes are merged into one making a single class. This type of coupling is the strongest as all fields/methods/properties of the class can be called. A god class will typically have low cohesion (which is explained in 1.3) as unrelated classes are merged into one.

Class - class bidirectional Two different classes which are dependent bidirectionally. This coupling is still very strong as a change in a single class might result in a change to the other class. This type of coupling is less strong compared to the god class as the communication is restrained to the public api (application programming interface) of the class.

Class - class unidirectional Two different classes with one class being dependent upon the other. A change to the server class (see 2.1) might lead to a change in the client class. The client class can be altered without fear of breaking the server class. This type of coupling is less strong compared to the bidirectional coupling because only changes to the server class potentially alter functionality of a different class.

Class - class through interface The api of the server class is abstracted by an interface. Changes to the server class only lead to changes in the client class if the definition of the interface changes. The interface hides the implementation details of the server class exposing only certain methods and/or properties. This type of coupling is considered less strong because of the added abstraction which hides the implementing class and its implementation details.

The following aspects of a dependency also affect the strength of coupling:

Number of interactions Two classes that are coupled but the amount of coupling is minimal (e.g. the client class calls only a single method on the server class) have a less strong coupling compared to two classes with a lot of interactions. Because the increased number of interactions it becomes more complex to separate the two classes making them coupled more strongly.

Scope of access The scope of the coupled member. A wider scope (e.g. a global attribute versus a local variable within a single method) has a longer life-cycle. This occurs as it goes out of scope later. The coupling is made stronger because it is available longer and to more methods.

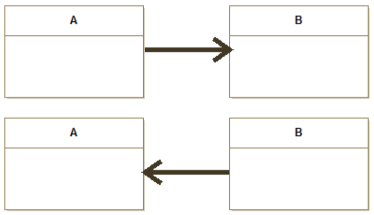

Figure 1.1: Viewed from class A, an example of efferent or import coupling at the top, afferent or export coupling at bottom.

Stability Defines the likelihood to change. This is apparent when one claims that coupling to the implementation is worse than coupling to the interface. Being coupled to lots of other classes and/or methods that are considered stable is less harmful as they will not change. Framework types like integer are considered stable; user-defined types are considered unstable. Therefore, coupling to a user-defined type is more harmful than coupling to a framework type.

1.2.2 Aspects of coupling

Two classes can be coupled to each other in various ways. An overview is listed below, in increasing order of malignity[10]

Data Classes communicate through scalar parameters.

Stamp A class contains a method that has a parameter of a different type.

Control Parameters are used to control the behaviour of the coupled class.

Common Classes that use the same global data.

Content Classes depend upon each others implementation details (e.g. reading a field of the other class after calling a method to read the result).

Coupling always has a direction which can be import (or efferent) or export (or afferent). Import and efferent coupling both mean the current type (for which we are calculating coupling) is dependent upon a different type. Export or afferent coupling is when a different type is dependent on the current type. See also figure 1.1. The direction is important as a developer can control the import coupling of a class.

1.3 Cohesion

Closely related to coupling is the concept of cohesion. Cohesion is a measure of how well various elements belong together. Cohesive classes can encapsulate [17] all behaviour related to a single problem hiding the details from consuming classes. Cohesion is lowered when unrelated features are added to a class. As with coupling, cohesion can be calculated both at the class and module level. An example of a metric for cohesion is the lack of cohesion in methods metric invented by Chidamber and Kemerer[5].
Cohesion is important as it is the antagonist of coupling. In the aforementioned case of a god class there is zero coupling (since there is only a single class) but cohesion is sacrificed as unrelated functionality is merged into a single class. On the opposite, separating every single method into its own class will provide a high value for cohesion. However, this comes at the cost of increased coupling as all these classes have to be connected in order to create a meaningful application.

1.4 Structure

This thesis is structured as follows, chapter 2 how coupling is related to maintenance. Chapter 3 describes the research method chosen followed by chapter 4 detailing the execution of the research. In chapter 5 the results are summarized which are analysed and discussed in chapter 6. Code samples are contained in the appendixes.

Chapter 2

Motivation

The level of coupling impacts the required maintenance effort both positively and negatively in multiple ways.

2.1 Clustering or decomposition

During the design process of a software system the entire system is decomposed into small parts of functionality that have as few inter relations as possible. This process is known as clustering, decomposition or modularization, the resulting groups of elements are called clusters, subsystems or modules. A single module provides a subset of the combined functionality provided by the entire system.

2.1.1 Functional cohesion

Modules within a system should have strong functional cohesion. This means all classes within the module are strongly related based on their function. A high amount of functional cohesion indicates the module addresses a single concern. This single concern is encapsulated within the module hiding the implementation details from the rest of the system. Separation of concerns and high functional cohesion have a positive impact on the maintenance of a software system[20].

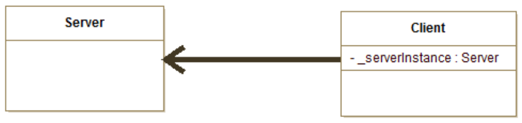

Figure 2.1: Example of a client class being coupled to the server class by a private field.

2.1.2 Reuse

By abstracting the functionality of a module to a level that is specific to the specific application, modules can be reused in a different context outside of the original system. As applications share some generic concerns (e.g. logging to a file) a module with the proper abstractions can be reused in different systems. Maintenance effort needed to maintain a module that is reused in multiple applications is less costly as the costs can be shared over all systems reusing the module.

2.1.3 Error-proneness

Selby and Basili have shown that modules that have a high coupling/strength ratio are more error prone compared to modules that are decomposed better[19]. As a result decomposing a system will result in less errors in the long-term making decomposition a form of preventive maintenance. Also, as the size of the module increases it becomes more error-prone, potentially to a point where further decomposition is needed.

2.2 Interfacing

Interfaces can be used to specify the contract for a module. By using interfaces the implementation can be separated from the functional definition. Interfaces are used frequently when lowering coupling as coupling to an interface is considered better compared to coupling to the implementation[10].

2.2.1 Functional description and program comprehension

An interface can provide a functional and abstract representation of a piece of code (i.e. class or module). Doing so, it hides the implementation details providing only a limited and high level description of the capabilities available. This high level functional description can be helpful to developers who are new to the system. Using the interfaces they can get a high level overview of the modules in the system; and which concern each module is addressing[21]. When performing maintenance on a system it is important that the system is fully understood to oversee the implications of a change. This holds for all types of maintenance.

2.2.2 Impact analysis

If a change is proposed, before the maintenance is performed, the impact of the change has to be determined. This impact analysis can be simplified when interfaces are used to abstract the various modules because the interfaces define the interactions for each module. Looking at the interfaces can tell, the developer, if the change will be contained in a single module or spreads through the application. A change that is contained within a small portion of the system is likely to be less costly to fix because less code is altered reducing the risk of introducing new defects.

2.2.3 Polymorphism and extensibility

Coupling to the implementation instead of the interface is considered worse as there can be multiple implementations of an interface. If there is a request to change the logging module to send mails instead of logging to a file a new implementation of the same interface can be created. Both implementations can now be used to handle the logging concern.
This concept is called polymorphism and adds flexibility[6] for varying implementations of a module to the system. This can be very useful for perfective or adaptive maintenance (e.g. the scenario above) as well as for extending an existing system (by configuration or inversion of control). Especially in application that provide only building blocks it is important to allow a consuming application to replace certain behaviours with its own if needed.

2.3 Testability

A large and complex system cannot be maintained properly without a set of automated tests. Testability is important as having tests can compensate for some deficiencies in for example the design of the application. A loosely coupled application aids testing.

2.3.1 Isolated tests

A decomposed system consists of multiple modules each handling their own concern. Because of this decomposition it is possible to write unit tests that validate the behaviour of a single module. There is no guarantee that the system as a whole will function correctly if all modules function correctly but the concern of a single module can be tested in isolation. Regression or integration tests can be written to verify the behaviour of the entire application.
A unit test is supposed to test a single functional requirement making it very focussed[1]. Using multiple tests for different kinds of input the output can be verified. These automated tests can be used as a safety net when performing any type of maintenance on a system. If one of the tests fail the corrective maintenance is targeted at a very small part of the system because of the small amount of code hit by the test. Finally, these tests can function as a description of how the module can be used as they test the same functionality that is exposed to other parts of the system. This way a test provides examples of the different ways the module can be used.

2.3.2 Mocking

As unit tests validate the correctness of a single function or requirement, they should only fail if this functionality is changed. However, decomposing a software system does not remove their inter-module dependencies. Depending only on interfaces of other modules makes the system flexible so polymorphism can be used.
Aside from a completely different implementation polymorphism can also be used for mocking. Mocking is a concept that replaces a dependency with a very simple implementation of the interface. This mocked implementation is very useful in testing as it makes tests more reliable as the other modules are replaced resulting in predictable outputs from these dependencies.

By using mocked implementations a failing test can only be caused by changes inside the tested module. Also, in some cases using mocked implementations can speed up the execution time of the test (e.g. by removing a dependency to a webservice). This makes them more likely to be run frequently as a quick result is provided.
During the development of a system mocking can be used to replace an unfinished module. This allows for parallel development which can reduce development time. When all modules are finished the mocked implementations can be discarded (or reused for testing purposes) and replaced by the now finished implementation.
Mocking does not lower the maintenance effort needed, it might even increase as the mocks need to be kept up-to-date, but it can be a very useful tool in supporting the testability of the system. This testability provides assurance to developers executing maintenance that no defects are introduced.

2.4 Downsides

2.4.1 Added complexity

Adding abstractions comes at a cost, the indirection that is added by the interfaces makes the control flow less obvious. By using interfaces the control flow is only visible at runtime because of dynamic dispatch [18]. In the case of the logging example, the developer will not know during compile time which implementation will be used (either the file or mail implementation). However, in theory the developer should not have to care about what implementation is used. He should limit his knowledge of the module to its interface, just making the call correctly, and not care about how his call is handled. This is the promise of proper encapsulation [17] which is lost if the developer is required to know the implementation details (which is also called a leaky abstraction).
Frequently a concept called 'Inversion of Control' or IoC is used to decompose the system [14]. While this system allows modules to be very loosely coupled, it adds complexity as well. Some developers might not be familiar with the principle making it more difficult for them to understand the system. Also, to use IoC a level of configuration is needed that should be maintained as well. The amount of additional effort needed varies with the tool used.

2.4.2 Additional code

To lower coupling abstractions have to be created. These abstractions need to be maintained just like other code. In extreme cases the amount of places where maintenance needs to be performed is doubled by decomposing a system. Lowering coupling by decomposing a system is a form of preventive maintenance that is beneficial in the long term. For small systems that are not likely to be reusable or maintained for long periods of time it might not be worthwhile.

2.4.3 Additional cloning possible

Because modules of the system are separated and become restricted in their interactions there is an increased risk of code cloning. Modules that are unrelated might still have some similar needs, e.g. parsing an input in a specific format, but this functionality has become unreachable because of decomposition. To 'reuse' the existing code a developer might copy the routine to the other module making a code clone. This is undesirable as code clones have proven to be a source of defects increasing the maintenance effort needed [11]. The proper solution is to have a module that exposes only these cross-cutting concerns, but this is not always possible.

2.5 Problem statement

Many metrics have been proposed in order to measure coupling effectively [3] but there is a lack of quantitative results that validate these metrics. This thesis aims at validating the impact of coupling on software maintainability. Having such data can help practitioners in making a decision about how much time and effort they are willing to invest in keeping their system loosely coupled.

2.5.1 Research Question

Because of the important relationship between coupling and cohesion, the research question of this thesis is defined as:

Does lowering coupling with unchanged cohesion ease mainte-

nance?

Exact type of coupling

Using the definition from Briand *et al.*[3], this thesis will focus only on direct import coupling. As a result indirect coupling and export coupling are not within the scope of this thesis. Direct import coupling counts the number of distinct classes a class is dependent upon.
Import coupling was chosen over export coupling as changing export coupling would require changes to other classes (the ones dependent on the class under maintenance). Import coupling however can be influenced within the scope of the class under maintenance making it more easy to influence. Also, classes that depend on the current class under maintenance might be in a different part of the system making it hard for a developer to easily oversee the consequences of removing the dependency. Finally, if import coupling is lowered the export coupling will automatically decrease (e.g. if the client class has an import coupling to the server class this server class has an export coupling to the client class). Direct coupling was chosen for the same reasons, it can be influenced directly when maintaining/developing a class.

Inheritance

Although inheritance is a form of coupling, as it provides access to another class's methods, it is not always considered harmful. As long as Liskovs substitution principle is obeyed[12], inheritance can be a very powerful tool that is important to good class design [13]. Determining if Liskovs substitution principle is obeyed is difficult, which makes it hard to qualify the coupling as being harmful or not. Therefore, inheritance is left outside of the scope of this thesis.

Cohesion

This thesis focusses on the influence of coupling. Because coupling and cohesion are not independent concepts the research will leave cohesion at its original level. If cohesion were to be altered too, it would become difficult to assess whether specific measurements are related to the lowering of coupling. Also, the possibilities for refactoring would be endless potentially resulting in very deep refactoring of the system under investigation. It is expected for a system to become more maintainable if deep refactoring is applied as that is the goal of refactoring. Maintaining the same level of cohesion allows the results of the research to be completely attributed to the effect of coupling.

Chapter 3

Research method

This thesis uses a research method that is based on the scientific method but the literature study was done first in order to come up with a proper research question. To break down the research question, "does lowering coupling with unchanged cohesion ease maintenance?", the following hypotheses are used:

- Maintenance becomes more localized because of reduced coupling.
- Reducing coupling can ease maintenance by making the system easier to understand.

These hypotheses are chosen because these are the places where it is most likely to find evidence of the influence of coupling. Cohesion levels will be kept at their original level, the hypotheses focus on the part of coupling that is related to the abstractions that have to be created to reduce coupling.

3.1 Approach

To answer the research question and (in)validate the hypotheses, a comparison between a system with low coupling and a system with a large amount of (strong) coupling is needed. Unfortunately it is unlikely there is a suitable system that has two revisions that differ only by their amounts of coupling. Because of this limitation, a system will have to be altered to lower the coupling. This in order to create a situation in which both systems expose the same functionality but do so using a different level of coupling in their

code-bases.
The research consists of the following phases:

- System selection
- Refactoring the existing system
- Applying maintenance scenarios to resulting systems and collecting data related to the maintenance.
- Analysis of the results

3.2 Phase 1: System selection

The first step is to select a proper system. Below is a list of the requirements needed for selecting a proper system and the rationale for these requirements.

3.2.1 At least 30.000 lines of code

Systems that are small are easier to maintain than large systems. Oman *et al.* [16] indicated that maintainability decreases when system size increases. This can be measured in lines of code (LOC) or by using the metrics provided by Halstead [9]. To prevent selection of a system that is too easy to maintain, a minimum size of 30KLOC was used. The metric of LOC is chosen over the Halstead metrics as LOC is easier to calculate and is often reported by online repositories. This makes selection based on LOC more suitable.

3.2.2 High amount of strong coupling

A system that is decomposed into modules and has a low amount of coupling is less suitable for this thesis. Results are expected to increase as the difference between the refactored system and the original system grows. Preferably the system will have no modules at all and contains a lot of internal dependencies. To assess whether a system matches these requirements, metrics on coupling will have to be calculated for the system. Based on these metrics a system can be selected that has the highest values for coupling.

3.2.3 Solid body of unit tests

Each refactoring session has the potential of breaking some part(s) of the system. Having a large set of unit tests makes it easy to test if there are any unforeseen consequences of the changes that were made. Furthermore, these unit tests provide a suite of integration tests that can be used to verify whether the systems' behaviour has not changed after refactoring.

3.2.4 Other considerations

There are some properties that the selected system preferably possesses.

The system is used actively.
A system with a large active userbase is preferred as this indicates that the product is mature.

Multiple versions of the system have been released.
A system that is being maintained for several versions is more likely to be more difficult to maintain than a new system. With each release, some features are added and others are altered. Some functionality has to be changed because of changed requirements. Some of these changes can be added easily as the system was designed to be flexible toward those changes [6]. Other changes are harder to implement and move the system away from its initial design, making future maintenance more difficult.

Java or C# based.
This is merely because these are both mature object-oriented languages. As a result, they have strong and integrated development environments in which to program. This is important as functionality provided by the IDE can ease maintenance, e.g. by making frequently occurring maintenance scenarios automated.

Risks

When the system has been selected, the specific version that is selected is of importance. Most open source systems also use pre-releases that are used for testing and by early adopters. These versions often contain bugs as development has not completed. Which version will be best will be hard to decide a priori. Therefore, it is not a requirement for the system.

3.3 Phase 2: Refactoring

After selecting the system, two separate versions of the system have to be created. The first version, or original version, of the system is the state of the system prior to any refactoring. In order to create the second version, or the refactored version, refactorings need to be applied to the system. As this thesis only focusses on the results of lowering coupling when cohesion is kept at the same level, only a subset of refactorings can be applied. Below are the refactorings, using the terminology from Fowler [8].

3.3.1 Extract interface

Extracting an interface is a refactoring that can be applied to a class. It creates a new interface that contains some (or all) of the public methods exposed by the class. Other classes can use the interface representation of the class instead of coupling directly to its implementation, lowering their coupling. Coupling to the implementation of the class is considered worse [10] than coupling to an interface representation of the class as the interface is likely to be more stable and it support polymorphism.
Using these newly created interfaces, modules can be isolated by providing only the abstraction (interface) to consuming classes. This means all classes that were coupled to the implementation will have to be changed to use the interface instead. Using interfaces makes it possible to make the implementing class completely unreachable from the client class; it is then accessible by its public interface only. This way it becomes impossible to be coupled to the implementation. However, this depends on the features of the programming language that is used.

3.3.2 Pull members up

Pulling members up moves a member to a more abstract level, which can be both a base (or super) class and an interface. This refactoring is sometimes needed after extracting an interface. When the interface is extracted, only public members are added to the interface. Languages like C# and Java, however, also support internal methods which can be accessed only from within the same project or package. When the consuming classes are changed to use the interface instead of the implementation, these methods are missing as these are not extracted into the interface at first as they are not public. By pulling members up, the members are added to the interface and made public (as is required by the interface).

3.3.3 Change bidirectional association to unidirectional

Bidirectional dependencies are undesirable [7] because they make software more complex. This is especially true for coupling because it is very hard to decouple two elements when they are mutually dependent upon one another. By making a dependency unidirectional decomposition is made possible.
This refactoring will not be applied frequently as it will involve deep refactoring of the code. This is undesirable as the level of cohesion needs to be maintained while lowering coupling.

3.3.4 Risks

The choice to leave cohesion outside the scope of this thesis can be considered a risk as it limits the level of refactoring. As stated before, coupling and cohesion are two concepts that are closely related. This choice is made because of the potential risk of altering the system too much, resulting in a better maintainability that cannot be attributed to the lowered values for coupling.

3.4 Phase 3: Applying maintenance scenarios to resulting systems and collecting data

Having an original and refactored version of the software gives the opportunity to do the same maintenance on both systems and collect information of the effort needed to make these changes. Some of the efforts done to maintain a system are easily measured (for example, amount of LOC changed); others are harder to measure as these are cognitive processes (how easy is it to understand the system).
During this phase, a number of maintenance scenarios, from the issue tracker of the selected system, and two custom scenarios will be executed. The issues from the issue tracker are used because these are objective and real maintenance scenarios. For an issue to qualify it has to, be an issue in the selected release and have a failing test. A failing test is important to ensure the fix was done properly, which should make the test pass. The custom

scenarios are used to illustrate the benefits of lowered coupling, these are not meant to be objective. The custom scenarios should be plausible maintenance scenarios meaning they are very likely to be executed in the future. While executing the scenarios, the following data will be collected from both systems:

- The number of lines changed.
- The number of files changed.
- What part of the system the changed files are in.
- How much time it takes to perform the maintenance from start to end.

3.4.1 Metrics

In order to get insight into how much the coupling has decreased, the refactored system is analysed and compared to the original system. This is done by calculating metrics (for direct import coupling and LOC) for both applications.

Response for Class

As a measure of control, values for Response For Class (RFC) will be collected as well. RFC is a metric that was invented by Chidamber and Kemerer[5] and is described as follows: "The set of all methods that can be invoked in response to a message of an instance of the class". It was proposed with the following viewpoints:

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester.
- The larger the number of methods that can be invoked from a class, the greater the complexity of the class.
- A worst case value for possible responses will assist in appropriate allocation of testing time

Based on the viewpoints this metric is well suited to be used as control values within this thesis. It contains both a viewpoint for understanding as well as for complexity. The validity of this metric was checked by Selby and Basili and found to be a good indicator of complexity and error proneness[19].
This metric decreases as a result of decoupling, however the amount by which it decreases is dependent upon the way of decoupling. Every public member that is only used within the module is not needed in the interface making classes coupled to the interface less coupled to the implementing class. If lowering the coupling allows for more members to be excluded from the interface this reduces the available methods to consuming classes, lowering the values for RFC.
Direct import coupling is a measure of the amount of types that are coupled to a specific class. It does not tell something about the influence of these classes on the complexity. RFC adds this value by saying something about the importance and relevance of the reduced coupling. For example, if import coupling was at 50 and it is lowered by 20% the decrease in RFC could be both 2% (the removed classes had very few methods) as well as 90% (the removed classes had a relatively high amount of methods). Finally, a decrease in RFC indicates lowered coupling is improving encapsulation of the isolated systems as fewer methods are exposed to the rest of the system. Therefore, it will be used as an objective check of how much the complexity is lowered and understanding is improved.

3.5 Phase 4: Analysis of the results

At the end of the experiment, the data from the experiment will be analysed to validate these hypotheses.

3.5.1 Maintenance become more localized

The results from the measurements specified in 3.4 indicate the location of the change. In order to (in)validate this hypothesis, the data from the experiment is analysed to see whether the necessary changes remain within a small part of the application. By using scenarios that stem from the issue tracker of the system, the relevancy of the used scenarios should be guaranteed and bias or influence on the scenarios minimized.
The issues are selected from the tracker after the refactorings had taken place. This is a potential risk as knowledge from how the system is refac-

tored is available. However, selecting the issues earlier (before refactoring) would allow the refactorings to be applied differently, making both options equally biased.
A system cannot be flexible for every change [6] making it unlikely every single scenario will become more localized. This hypotheses can be considered validated if half of the scenarios provide data that changes are more localized.

3.5.2 Reduced coupling can ease maintenance by making the system easier to understand

At the end of the refactoring phase, it is expected that the refactored system is decomposed into several smaller modules with clear boundaries described by interfaces. The isolation and clear responsibilities for a module should make it easier to understand.
This hypothesis can be validated by trying to write tests that run in isolation. This is a sign the decomposition can be used to isolate a concern of the system into a module that can be tested and maintained independently.
Also, this will be checked by analysing the values gathered from the RFC metric. A decrease of this metric indicates the isolation is an indication of increased encapsulation and decreased complexity maintaining a class as fewer members are exposed and need to be understood.

Chapter 4

Research

This chapter gives an overview of research described in chapter 3 and (unforeseen) challenges faced during the research.

4.1 Phase 1: System selection

From the start of the selection phase it was apparent that a candidate would fit the criteria very well: NHibernate. This system is known to be a good candidate because one of the developers explicitly stated that low coupling was not a goal they were trying to achieve[1]. A quick assesment of the code of NHibernate confirmed it matched all requirements. The metrics that were collected can be found in appendix A.
A quick scan of Ohloh[2] was performed to see whether other projects matched the requirements better than NHibernate did, but none were found. Because the developers of NHibernate are not aiming for low coupling, they made a perfect candidate. As a result, the first 'General Availability' release of the 3.0 version of NHibernate was selected[3] [4].

[1]Among others, see: `http://ayende.com/blog/4072/answering-to-nhibernate-codebase-quality-criticism`

[2]`http://www.ohloh.net`

[3]Dashboard: `https://nhibernate.jira.com/browse/NH/fixforversion/10350`

[4]Download: `https://github.com/nhibernate/nhibernate-core/zipball/3.0.0GA`

4.1.1 About NHibernate

NHibernate is an object relation mapper (OR/m). This is meant to bridge the gap between the relation database model and the object model in the application. It is used to translate queries on the object model into queries for a specific database management system and return the result as instances of one or more classes.
NHibernate originally is a C# port of Hibernate, which is an OR/m for Java. NHibernate has seen a number of major releases starting from November 2007 and is used actively by numerous projects.
The 3.0.0GA version of NHibernate was selected for this thesis because it is a so-called General Availability release. This is a final release after three alpha and two beta releases. A final release was selected as this is a version on which development has finished. An alpha or beta release would be less suitable as some features might not be completely finished.
In the 3.0 release of NHibernate the project was switched to .Net framework version 3.5 and along with it a Linq provider was implemented. Linq provides an abstract way to query over a collection of elements independent of it being an in-memory list, a database or a web service. This linq provider is built on top of the existing Domain Specific Language (DSL) for querying with NHibernate. This DSL is called Hql and can be used to create queries in string format that are interpreted at runtime to execute a query.
The codebase of NHibernate is made up of a single project of 67KLOC that contains all the code. On top of this assembly a DomainModel project is created which contains classes that are used for unit testing the project. The unit tests are contained in yet another project that uses the objects from the DomainModel project and tests the main project.
In NHibernate everything is contained in a single project but internally a limited amount of interfaces are being used. These interfaces are used to support polymorphism and extensibility for people using it as an OR/m framework, but are not used with lowered coupling in mind. As a result most of these interfaces are very large, describing many methods and properties making coupling through these interfaces strong even though it is using an interface.

4.1.2 Preparation

Before the next phase could start, the system had to be prepared for refactoring. A few small adjustments were made to make the project compile and pass all the tests that were in the project. Below are the adjustments that were made before the initial check-in on the public repository[5].

- Mark NHibernate assembly as CLS compliant (a test checks this and fails)
- Disable test NHibernate.Test.Linq.LinqQuerySamples.DLinqJoin5b as it fails
- Disable test NHibernate.Test.NHSpecificTest.NH1689.SampleTest as it fails

After these changes, the project compiles and all tests succeed. This version of the software will be our 'original' version.

4.2 Phase 2: Refactoring

Starting from the original version from phase 1, a new branch is created[6]. This branch will contain all the refactorings that are applied to reduce the level of coupling.

4.2.1 Creation of abstractions project

The first step in refactoring was to create a new namespace called NHibernate.Abstractions. This namespace was extracted later to an isolated module. But, in order to be able to take small steps at a time a separate namespace was chosen first. All code elements that are used for communication between modules should be placed in this namespace/module. While moving items to the new namespace, the tests were run frequently to assure the changes did not break anything.
Extracting an interface from a class and using this instead should not break anything (as long as there is a single class implementing the interface). However, unit tests failed multiple times during the extraction of the interfaces. These failures were caused by the use of reflection to look up a specific class. After altering the configuration files that contained the names of the classes, tests succeeded again.

[5]`https://github.com/thesis2012/thesis-nhibernate`
[6]`https://github.com/thesis2012/thesis-nhibernate/tree/refactor`

After extracting a number of core classes and interfaces to the abstractions namespace, the module was isolated from the main project to a new project. This module will contain the shared interfaces, some abstract base classes, enumerations and other elements used for intermodule communication. Only the interfaces that are used or exposed by other modules are moved from the main project to this new module. This results in the module growing as other modules are isolated from the main project.
When an interface is moved to the abstractions module, all types used in the interface have to be in the abstractions project as well. This is needed as the abstractions module is not allowed to reference the main project as this would break the isolation. As a result, references to classes defined in the main project will not compile. This means that for concrete classes used in the public methods of a class, an interface has to be extracted as well.

4.2.2 Candidate module identification

With the abstractions modules in place, other modules that can be isolated had to be identified. To do this, static analysis was used to see which parts of the system belong together, this selection is based on cohesion[4]. A module can be identified by a set of elements that are coupled very strongly but have little references to other parts of the system. This can also be described as having low coupling to other parts of the system and being very cohesive, making it strongly coupled, internally. This internal coupling is not a problem if the classes are highly cohesive[15][7]. By extracting a module, these cohesive classes encapsulate a single concern of the application, exposing only a limited set of functionality to the rest of the application described in the interface.
As an example, the Hql namespace has a lot of dependencies pointing towards it but is using very little of the rest of the system. This is an indication it can be isolated in its own module and be referenced from the main project. This way this module is isolated and abstracted to interfaces describing its functionality rather than coupling directly to the implementations within.

4.2.3 Extracting modules

Extracting modules out of the main project is an iterative process. It consists roughly of the following phases:

- Identification of module by static analysis focussed on finding cohesive sets of classes
- Creation of abstractions for elements in main project, coupled to one of the classes in the set, if needed and replacing the coupling to the class with the interface
- Extraction of the set of classes to an isolated module (project or package)
- Ensure successful execution of all unit tests, fixing broken ones when needed

This is an iterative process because, with each extracted module, new abstractions are created for coupled elements of the new module. This in turn might decrease the coupling of other modules that are still in the main project.
The following modules were extracted in chronological order:

Types Types used both in Sql and in Code and mapping between those.

Util Utility classes with generic functions for arrays etc.

Sql Code for communicating with persistent storage.

Linq Linq provider.

Hql Hql Abstract Syntax Tree (AST) and logic.

Cache All kinds of caching.

Event Hooks to certain events a consumer can hook into (e.g. upon saving of an object).

The new projects for Types and Util are more library-like projects and do not have a function of their own. As a result, these projects are referenced more frequently compared to the others. An overview of the references between the new projects is given in 5.1.2.
After isolation of the modules, the interfaces and abstract classes in the abstractions modules were stripped from all unused methods, parameters and properties. This was done to reduce the interface to the minimal set of functionality that is needed. It is important for the interfaces to have as

few members as possible. This is because an increased amount of elements increases the public API that is supposed to remain stable and increases the strength of coupling. Having a very rich interface constrains the development of the abstracted module because changing the interface is undesirable as the other modules depend on this interface. Unfortunately, the interfaces remained quite big. Members of the interface could only be safely removed if they were not used because consuming classes were not altered as this might alter the level of cohesion.
Ultimately the main project was reduced to 40% of its original size. Preferably it had been shrunk even more by extracting more modules. However, the remaining code is coupled so strongly (most of it bidirectional) making it impossible to isolate additional modules unless more rigorous refactoring is applied.

4.2.4 Difficulties

Because NHibernate is not built with low coupling in mind, some difficulties were encountered while refactoring.

Constructors

The biggest challenge were the constructors of classes in the system. When interfaces are extracted, all methods and properties can be added to the interface except the constructors. As a result, the calls to a constructor of a type for which a new interface was extracted had to be replaced by something else that returns a new instance of the interface without exposing the concrete class underneath.
There are multiple ways to achieve such behaviour, but they vary in their impact on the code. The most elegant solution would be to use a concept called inversion of control (or IoC) or Dependency Inversion Principle [14]. These concepts alleviate the requirement for calling the constructor by configuring a new object called the container. This container is a factory containing registrations of an interface and a concrete class to be used. From the code, a new instance of a specific interface can be requested and the container will construct a new instance of the configured class. As an extra benefit, it can supply instances for the dependencies (arguments of the constructor) of that class when constructing it, it is configured to return instances for those interfaces as well.

Although it is the most elegant solution, the introduction of an IoC container would impact the code too much as dependencies would have to be declared in the constructor. Instead of a full-fledged IoC container, a factory class was implemented. This new class contains methods which are configured to point to a specific constructor during application startup. This way the constructor calls can be replaced by a call to one of the methods of the static factory. This way the class becomes a factory for various types.

Bootstrapping

With the introduction of the factory, it became necessary to add code to the application that is run at start-up to configure the static factory appropriately. As the system now consists of multiple projects, this becomes challenging. In order for all methods in the static factory to be initialized, all projects have to be scanned. To make this more easy, an interface is declared. All assemblies in the system are scanned for classes deriving of this interface. If the class is found, the set up method (of the interface) is called. This gives each project the opportunity to configure its own methods on the static factory. For example, the Cache project initializes the method for instantiating an instance of the ICacheKey interface.

4.3 Phase 3: Applying maintenance scenarios to resulting systems and collecting data

During the third phase, the maintenance associated with the selected maintenance scenarios was carried out on both the original and refactored version of the system. After finishing the maintenance, metrics were collected to compare the two versions on their metric values.

4.3.1 Issues

A total of twelve issues were found matching the criteria for selection. These issues were taken from the public issue tracker[7] of the project. Selecting issues was difficult because the issue tracker allows you to filter the issues by the 'Affects version' field but this has proven to be incorrect in a few

[7]`https://nhibernate.jira.com`

Table 4.1: Issues from tracker

Issue #	Type	Priority	Resolution	Failing test	First
2203	Bug	Minor	Fixed	Yes	After
2362	Bug	Major	Fixed	Yes	After
2400	Bug	Minor	Fixed	Yes	After
2433	Bug	Major	Fixed	Yes	Before
2452	Bug	Critical	Fixed	No	Before
2473	Patch	Minor	Fixed	Yes	Before
2490	Bug	Minor	Fixed	Yes	After
2507	Bug	Major	Fixed	Yes	Before
2549	Bug	Trivial	Fixed	Yes	Before
2559	Bug	Critical	Fixed	Yes	Before
2649	Bug	Critical	Unresolved	Yes	After
2913	Bug	Critical	Fixed	Yes	After

cases.
In the 3.0GA version of the software very little issues have been reported, so selection was expanded to issues that are reported for the 3.1 version. To qualify, the issues had to have a failing test to check the issue (and fix). This allowed issues reported for version 3.1 to be used as well as long as they had a test that also failed in the 3.0GA version. As a result, the selection was done based on issues that were fixed in the 3.0GA or the 3.1 release. Also, because the Linq provider was the major new feature for the 3.0 release, a lot of the issues reported are directly related to the Linq provider. Unfortunately all of the issues can be classified as corrective maintenance.
Of the total of twelve issues, three could not be fixed because they were related to the parsing of Linq expression trees. The parsing was handled in a separate, external assembly. Table 4.1 lists the issues covered in the maintenance scenarios. The last three rows contain the issues related to the external Linq parser.

In table 4.1, the column to the right describes the version of the system the issue will be fixed in first. Determining which version was fixed first was done based on the priority. Both versions should have the same amount of high priority issues when possible. This was done because it is impossible to not be biased when fixing the issue for the second time in a different version of the same system.

4.3.2 Custom scenarios

Two custom scenarios were selected to be added to the maintenance phase. These scenarios were used to provide insight into what flexibility is acquired by reducing coupling [6]. These scenarios were selected after the refactorings had taken place. This was done to ensure the scenarios provided the desired insight. The first scenario is to implement the 'having' statement in Hql while the second is about the opportunities for testing in isolation. Both are explained below.

Implement proper handling of 'having'

The first scenario is adding proper support for the 'having' statement in Linq and Hql. Having is a predicate that can be applied when aggregating results in SQL. In the original version of the software, the having construct was handled by using a 'where' statement which is incomplete because of the different semantics. The lack of proper having support became apparent when issue 2452 was fixed because this fix caused another test to fail. During the maintenance on the issues, this test was temporarily disabled to have passing tests, because this was to be fixed in a custom scenario.
This scenario was selected because it requires changes to the Abstract Syntax Tree produced by both Linq and HQL. This AST is an intermediate model describing the query that can be translated into another model (i.e. a sql string for a specific dialect). Because other parts of the system are built on these AST, there is potential for the ripple effect [15]. Finally, this scenario is a useful addition to the system that is expected to occur in the future.

Use mocking for isolating tests

The second scenario focuses on the new capabilities that come with the looser coupling by adding the abstractions. During this scenario, functionality that is outside the scope of the test should be isolated by using existing mocking tools. This scenario was chosen as it should be a good example of the flexibility that is gained when lowering coupling. Two cases were added to illustrate the benefits of the decoupled system. The Moq[8] library was used for our mocking needs.

[8]http://code.google.com/p/moq/

The first case was a problem that occurred when fixing issues. There was one test (related to issue number 2400) that seemed troublesome; it failed when all tests in the class were run but succeeded on its own. Investigating the failure pointed to a cache for query plans. These query plans are used to create query only once and caching it afterwards. This is beneficial for performance of the running system. However, it is unwanted in this test because it adds side effects to our running tests which is undesirable. In order to get this test to work properly, we mocked the behaviour of the cache to always return a new QueryPlan, effectively disabling the cache, so it works independently of the other tests.

The second case is an example of a module in complete isolation, in this case the Linq project. Since a lot of the issues that needed to be fixed were in the Linq project, a way to test these issues in isolation was needed. As most of the issues related to the Linq project could be checked by testing the AST that was produced, an attempt was made to isolate the construction of the AST.
In order to get the AST, an instance of the ISessionFactoryImplementor interface was needed while only one method and one property was needed. After creating a mock implementation of the interface, the original method for getting the AST was called supplying the mocked instance as a parameter.

4.3.3 Metrics

To make a good comparison of the two versions metrics need to be collected for both versions of the system. The LOC metric can easily be obtained as numerous external applications are capable of doing so, For this thesis the code metrics viewer provided by Microsoft was used.
For coupling and RFC, no proper tool exists that does exactly the calculation the way it was specified in the research question (or it is not explicitly stated how it is calculated). So, for calculating direct import coupling and response for class, a simple program that calculates these metrics had to be written.
Fortunately this can be done by using the new Roslyn project of Microsoft[9]. Using this software, new metrics for coupling and RFC were built on top of the C# parse tree that is generated by the compiler. Because the parse tree is completely built by this tool, only the calculation based on this parse tree

[9]http://msdn.microsoft.com/en-us/vstudio/hh543936

had to be built. The quality of the parse tree is guaranteed as this parse tree is used internally by the C# compiler.
To analyse the generated parse tree Roslyn provides a rich set of assemblies that wrap common scenarios. For analysing the parse tree Roslyn uses the visitor pattern. A class can be derived from `SyntaxWalker` which has a virtual method for every possible node in the parse tree. By overriding these methods, code can be added that handles the found node. For calculating the RFC and coupling metric the methods for `ClassDeclaration` and `InterfaceDeclaration` had to be overridden. As the name suggest these methods are called for every class or interface that is encountered within the parse tree. The metrics obtained were verified by comparing metrics calculated by hand with the results from the tool for a random set of 10 classes. Below is the detailed specification of both metrics, code for both classes can be found in appendix C.

Direct import coupling

A class deriving from `SyntaxWalker` was implemented with an overridden `VisitClassDeclaration` and `VisitInterfaceDeclaration` method. Inside these methods the types of the following elements, of the interface or class, were selected: properties, fields, method parameters, constructor parameters and local variables of methods. From this list of types, the types that have a name that starts with NHibernate and are not equal to the visited class declaration were selected. External dependencies are not considered as these can be considered to be stable from the point of view of NHibernate (the code is not part of NHibernate). These external dependencies can only change if they are updated to a newer version. The resulting types are put in two dictionaries: one for classes and one for interfaces. This dictionary allows a value to be looked up by a key, in both cases the key is the fully qualified name of the class or interface.

Response For Class

The response for class metric is implemented as class deriving from `SyntaxWalker` with an overridden `VisitClassDeclaration` and `VisitInterfaceDeclaration` method. This walker generates a dictionary which contains the number of public methods and/or properties per type. This dictionary uses the fully qualified name of the class or interface as key and the metric value as value.

Using the coupled types from the coupling metric, the RFC metric can be calculated by summing the numbers from the RFC metric for all classes to which a class is coupled.

Results

The results of the metrics were written to a csv file so they can be analysed using an external program.

Chapter 5

Results

This chapter contains the results of the research, no code examples are included these can be found in the public repository[1].

5.1 Isolation

As a result of the refactorings the system which consisted of one big module has been decomposed into 9 modules. Of these 9, 5 can be seen as isolated modules with a specific responsibility being Caching, Events, Hql Parser, Linq provider and Database communication. Three of the remaining projects, Abstractions, Types and Util can be seen as isolated library projects that contain no important logic. Functionality within these projects describes various types supported by databases (Types), functions used for merging arrays and concatenating strings etcetera (Util) and the interfaces and enumerations used for intermodule communication (Abstractions). The remaining project are the remnants of the original project.

5.1.1 Lines of Code

The extraction of proper modules resulted in a significant decrease in the amount of LOC in the main project. This can be illustrated by the following table listing the LOC metrics obtained on both the original and refactored

[1]`https://github.com/thesis2012/thesis-nhibernate`

Before		**After**	
Project	**LOC**	**Project**	**LOC**
Nhibernate	67453	Nhibernate	24661
		Nhibernate.Abstractions	3271
		Nhibernate.Cache	684
		Nhibernate.Event	2621
		Nhibernate.Hql	19449
		Nhibernate.Linq	1921
		Nhibernate.Sql	7937
		Nhibernate.Types	5788
		Nhibernate.Util	1233
Total	**67453**	**Total**	**67562**

Table 5.1: Lines of Code metric data

system. As we can see from table 5.1 the number of projects has increased but this has a limited effect on amount of LOC. The metric used counts only lines of code in the body of methods, as an interface does not specify a body for its methods it is not counted. This explains the small difference in lines of code while numerous interfaces have been added to the system. However, the added interfaces have to be maintained as well. This will add additional effort when maintaining the system but keeping this in sync is a matter of updating the signature of a method in n+1 (where n is the number of implementations of the interface) places. Also, failure to update these interfaces will result in compiler errors making it impossible to forget as the system will not compile.

5.1.2 References

These new modules are very limited in their references making them isolated from the rest of the application. The references still needed are listed in table 5.2 and visualized in 5.1.
From table 5.2 it is clear that the isolated modules are referencing the library projects but not each other. The modules that were isolated but have a library function are used by virtually every other module. The abstractions project is referenced by every single module which was to be expected given that it contains all the interfaces that describe intermodule communication.

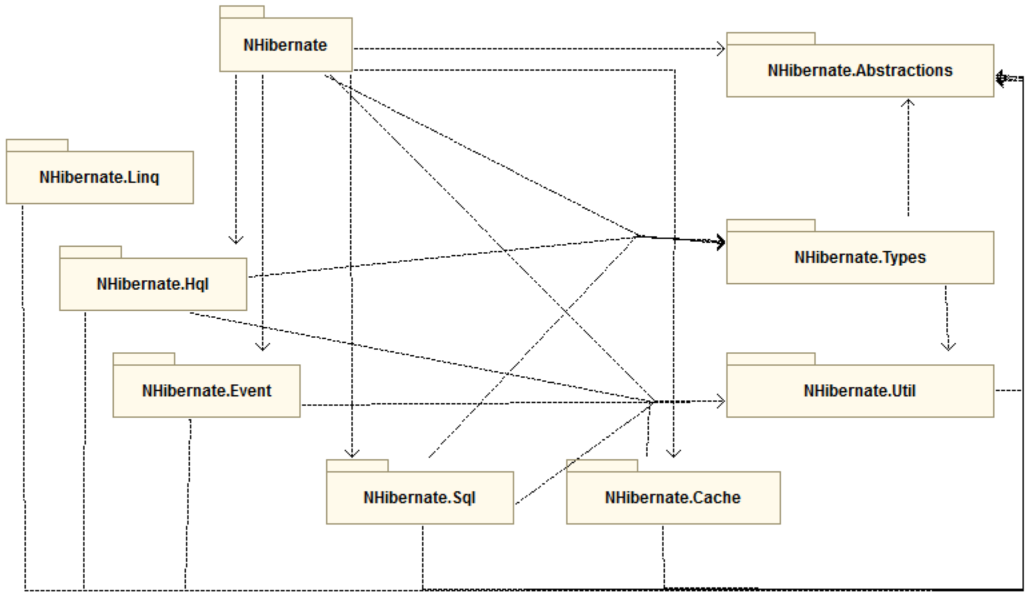

Figure 5.1: An image of the dependencies between projects after refactoring, before refactoring there was just a single project.

5.2 Issues

As stated in 4.3 of the 12 selected issues only 9 were caused by the system itself meaning they could be fixed. The other 3 were related to the external Linq parser. All 9 issues can be qualified as corrective maintenance which is unfortunate as it would have been better if at least one adaptive or perfective issue was included. The metrics gathered from the issues are reported in 4 tables, one for each measurement.

From table 5.3 and 5.4 we can conclude there is only a very small difference between the original and the refactored system for both LOC and files hit. This small difference can be explained by the added interfaces. If methods need to be added to the interface or the signature of one of the methods on the interface is changed there are two files to maintain instead of one. From this data we can conclude that only a single issue had an impact that might have consequences outside a module (as it changes the interface).

For the time taken we see very large differences in table 5.5. Seven out of nine issues were easy to fix with a time to locate and fix of two hours or

References	
Project	**Referenced projects**
Nhibernate	All
Abstractions	None
Cache	Abstractions, Util
Event	Abstractions, Util
Hql	Abstractions, Types, Util
Linq	Abstractions
Sql	Abstractions, Types, Util
Types	Abstractions, Util
Util	Abstractions

Table 5.2: References between projects

lower. Some issues show very big differences between the before and after system, this is caused by the knowledge about the issue when fixing it for the second time.
Two issues, i.e. 2452 and 2400, took a long time to fix because of the nature of the test. For example 2452 fires a very complex query and checks the result. Eventually it was related to the way the groupby statement was processed but this was not checked by the test. As a result everything, from parsing the query to the execution of SQL and returning the results could be the problem. A targeted test would have made fixing this issue much easier.

The final measurement is about how localized a change is. Obviously, in the before system with only a single module every change is contained within this single module. From table 5.6 we can conclude that in the after system $\frac{2}{3}$ of the issues are completely contained within an isolated module. This is beneficial as this means the change is contained within boundaries of this isolated module. This means a maintainer is not required to understand other modules for fixing this issue.

5.3 Custom scenarios

After finishing the maintenance on the issues the custom scenarios were executed. Below are the results of these scenarios.

Issue #	# Files before	# Files after
2203	2	2
2362	4	6
2400	2	2
2433	1	1
2452	2	2
2473	1	1
2490	1	1
2507	1	1
2549	1	1
2559	no data	no data
2649	no data	no data
2913	no data	no data

Table 5.3: Changed number of files in before and after system

Implement proper handling of having

The same data was gathered while implementing having as was gathered when the issues were fixed. The refactored system was altered first resulting in a very quick fix for the original system. The data collected from this scenario is summarized in 5.7.
Based on table 5.7, no ripple effect can be seen. The amount of LOC and files changed is too small. It does show that the addition of having is not contained within a single module in the refactored system. In the refactored version the changes are mostly within the Linq system. Changes to the Hql module were abstracted by the IHqlTreeBuilder and IHqlTreeNode interfaces. This explains the difference in files hit as these two additional interfaces had to be adjusted (which were both in the abstractions project). For this scenario, the lowered coupling does not influence the maintenance in a positive manner. It even adds additional interfaces to maintain, however the amount of time needed for this additional maintenance was minimal compared to the complete change needed.

Use mocking for isolating tests

The first case, mocking the query plan cache, was proven to be possible by using mocked objects. Because this test is testing the entire system by

Issue #	**# LOC before**	**# LOC after**
2203	7	7
2362	22	23
2400	20	20
2433	4	5
2452	29	29
2473	11	11
2490	4	4
2507	2	2
2549	1	1
2559	no data	no data
2649	no data	no data
2913	no data	no data

Table 5.4: Changed number of LOC in before and after system

executing a Linq query and checking the result, a considerable amount of mocking code had to be written. This was caused by the way the instance was retrieved, the classes involved can be seen in 5.2. To query we need an instance of the Northwind class, this is the code representation of a database with all elements we can query. This class creates a session through the OpenSession method and contains the QueryPlanCache property that needs to be mocked.
To mock this the ISessionFactoryImplementor had to be mocked. For this scenario only the OpenSession method and QueryPlanCache property had to be mocked. The OpenSession method is mocked as this is used by the Northwind class but this is not relevant for the behaviour we want to mock. The QueryPlanCache property is mocked by injecting a mocked instance of the IQueryPlanCache interface. This mocked instance is set up so it always returns a new IQueryExpressionPlan, effectively disabling the cache.
This may seem like a lot of work to disable a single cache. However, mocking the ISessionFactoryImplementor can be done once and reused over various tests. This reduces the effort of using mocked implementations.

The second case needed far less mocking code in order to be able to test the constructed Linq AST in isolation. A total of 4 lines were enough to mock all the functions that were needed to successfully construct a Linq AST. Because this test runs in isolation of the rest of the system only the functionality constructing the AST is tested. As with the first case, this

Issue #	Time taken before	Time taken after
2203	.25 hr	1.5hr
2362	.25 hr	2hr
2400	.25 hr	4hr
2433	.25 hr	.5hr
2452	9hr	.5hr
2473	2hr	.5hr
2490	.25hr	.75hr
2507	.5hr	.25hr
2549	.25 hr	.25hr
2559	no data	no data
2649	no data	no data
2913	no data	no data

Table 5.5: Time taken for applying fix in before and after system

mocking code is written once and can be reused for all tests that need to test the AST.

5.4 Metrics

The metric data from the original and refactored system can be found in A. From this metric data we can see that class coupling has decreased while interface coupling has increased. This was to be expected as the refactorings used aim at introducing interfaces to separate implementation from behaviour. Generally speaking we just see a shift from classes to interfaces which should be beneficial [10] but the total amount of coupled classes and interfaces has only decreased minimally. This might have occurred when minimizing the interfaces to the minimum possible.

The RFC metric has decreased but not by significant amounts. The average of RFC in the original system was just 67.7 compared to just over 55.8 in the new system. This decrease was expected as the abstractions provide less methods to the consuming classes compared to the implementing classes.
The original system had a small advantage in these metric calculations as only public methods were counted as a method that could potentially be called. In the original system some communication was done through internal methods that were made public in the after system so the effect is likely

Issue #	**% of LOC in module**	**% of LOC in module**
2203	100% Nhibernate	100% in Linq
2362	100% Nhibernate	100% in Linq
2400	100% Nhibernate	100% in Nhibernate
2433	100% Nhibernate	100% in Linq
2452	100% Nhibernate	100% in Linq
2473	100% Nhibernate	100% Nhibernate
2490	100% Nhibernate	100% in Sql
2507	100% Nhibernate	100% in Linq
2549	100% Nhibernate	100% Nhibernate
2559	no data	no data
2649	no data	no data
2913	no data	no data

Table 5.6: Location of changed LOC in before and after system

Table 5.7: Introduce having results

Revision	**# Files hit**	**# LOC hit**	**Time**	**% of LOC in module**
Before	6	55	0.5hr	100% Nhibernate
After	8	59	2hr	71% Linq, 7% Abstractions, 22% Hql

greater than the metric suggests.

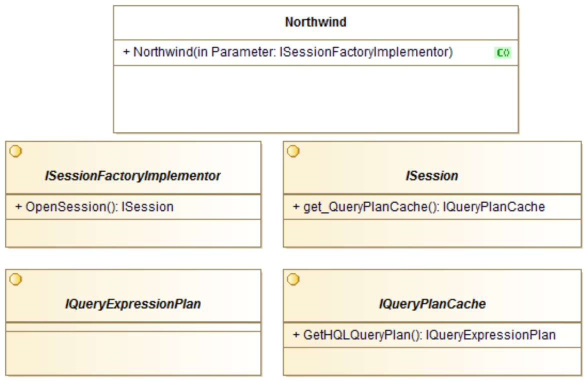
Northwind
+ Northwind(in Parameter: ISessionFactoryImplementor)
ISessionFactoryImplementor
+ OpenSession(): ISession
ISession
+ get_QueryPlanCache(): IQueryPlanCache
IQueryExpressionPlan
IQueryPlanCache
+ GetHQLQueryPlan(): IQueryExpressionPlan

Figure 5.2: Classes and interfaces involved in mocking QueryPlanCache.

Chapter 6

Analysis and Conclusions

Analysing all data gathered from our research leads to the following conclusions regarding our hypotheses.

6.1 Hypotheses

6.1.1 Maintenance becomes more localized

The data collected from our experiment does not show that changes become more localized based on the amount of lines and amount of files that need to be changed. It does show that changes can be kept within a small module when modules are isolated. This is supported by the values of RFC as well, they show only a small decrease making the 'ripple effect' only slightly less likely.
The abstractions did not help in finding the place where the code needed to be fixed because the tests were all written as queries and asserting the result, which does not target the search as the entire system is hit by the test. The results from mocking show that it is possible in the refactored system to write targeted tests that test little code which will make it easier to find the place where the issue can be fixed.

There are two factors that have a major influence on these results, the selected version of the system and the selected maintenance scenarios. It is possible that the selected issues would have had a different impact if the selected system was not a final release. After several pre-releases chances are

that the major issues, that need a lot of work, have been found and fixed. All the issues that have come from the issue tracker involved corrective maintenance and were very localized in both versions.
The fact that the major feature for this release of NHibernate was a new Linq provider might have influenced the results as well. Because the Linq provider is based upon abstractions supplied by the language (i.e. the IQueryable interface) this was already a part of the system that was relatively loosely coupled. As can be seen from the references in 5.1.2, this module could be isolated from the other modules completely. Because this was the major addition to the system it is likely for the issues to be in this particular area (which is supported by the data collected).

6.1.2 Lowering coupling can ease maintenance by making the system easier to understand

The increase in isolation helps in maintaining the entire system. The newly isolated modules have a single concern they are addressing, and clear boundaries for communicating with other parts of the system. As a result it becomes more easy to assess if a change is likely to be contained within an isolated module.
Isolating the modules has split the system into smaller, more cohesive blocks of code. Because these modules address a single concern, they are more easy to understand as a developer is not required to understand all modules at once. A new developer could start maintaining a small isolated (less critical) module first and be introduced to the entire system gradually.

The interactions between a module and the rest of the system has become more complete as internal methods had to be made public to be declared in the interface. In the refactored system the interfaces were minimized based on the refactorings available. The effect of this reduction can be seen in the metrics gathered for RFC. Overall a decrease of 18% was measured. Although this is not a very big decrease, the amount is considerable given the refactorings performed were pretty safe (as methods were not changed internally because of cohesion). Altering cohesion could have made the results better as the interfaces could have been reduced further. However, this would also increase the risk of introducing defects considerably making a good set of tests a requirement.

The created abstractions can be of help in testing as the modules can now be

tested in isolation. This way, more specific tests can be written that focus on the validation of a small part of one module compared to executing a query and checking the results requiring the entire system to work correctly. This can make the test code more explicit about what is being tested. This makes the test an example of how a module can be used as well as targeting the maintenance in case of a failure. The small decrease in RFC is not a problem for testability, the introduction of interfaces allows for the use of mocks that always provide expected results.
Also, because the mocks can replace slow dependencies, like webservices, the execution time can be decreased. This makes the tests more likely to be run frequently.
Finally, it is important to note that most lines of mocking code reported in the results have only to be written once for multiple tests. This minimizes the needed effort when more tests are written using this mocking technique.

6.2 Research question

The research question can only partially be answered. The results from this thesis indicate lowering coupling is beneficial to the overall system but it should be seen as a form of preventive maintenance. The results show no influence of lowered coupling on corrective maintenance of existing issues.
Lowering coupling enables the isolation of small and cohesive modules that can be maintained independently. This is easier because of the single concern and the clear communication with other parts of the system described by the interfaces. This effect was validated by the decrease measured in the RFC metric. The added possibility of using mocked implementations during testing and development (in case of an unfinished dependency) can be very valuable.
The impact from maintenance scenarios was not big enough to see a benefit of coupling as changes were very local in the original system as well. Because we did not see the ripple effect in the original system the required maintenance effort is even slightly increased with lowered coupling because of additional interfaces. If (non corrective) issues with a larger impact would have been available the effect might have been more visible but this is unsure.

Chapter 7

Discussion

There are a few things that could have been done differently in order to make this research more interesting. In hindsight the choice of the final version and, more importantly, version 3.0 has limited the available issues from the issue tracker. More care should have been taken when selecting this version, especially looking at the issues that were reported for this version which are small both in number and impact.
In version 3.0 the major addition was the implementation of the Linq provider which is practically always loosely coupled as the interfaces needed to implement this provider are supplied by the language itself.

Leaving cohesion out of the research is too limiting on the choice of refactorings. Because the internals of the code had to remain the same the refactored system did not benefit of the full effects of low coupling. Leaving out cohesion has proven to be a too safe choice. Because of this restraint the software could not be modified to limit the interfaces to the absolute minimum needed, which is supported by the values for RFC.

Finally, from the metrics gathered one can conclude that the current state of affairs of NHibernate is a system that is very difficult to maintain. However, they do manage to deliver updates and add new functionalities while keeping the old functionality intact. Which means there must be a way to maintain such a system. An important part of this is handled by the unit tests. Because there are so many tests checking all kinds of scenarios one can be pretty sure they deliver a product that has a certain level of functionality if all unit tests succeed. This is a radically different approach to maintainability but for this specific project it seems to work.

7.1 Future work

There are two research directions that extend this thesis. The research could be repeated when taking cohesion into account. Or it could be repeated using a different version of NHibernate which is more suited based on added functionality and issues reported.
Taking cohesion into account would open up more possibilities to really create the correct abstractions that are stripped to the minimal needed set. This would pose a challenge as to what level refactoring will be allowed to prevent from restructuring the entire application.
By selecting a different version, there is an opportunity to take a version that not just adds new functionality but also contains some redesign internally. It would be interesting to see if a redesign of existing functionality would be better supported by a system with lowered coupling.

Bibliography

[1] Beck, K. Embracing change with extreme programming. *Computer 32*, 10 (Oct. 1999), 70–77.

[2] Boehm, B. W. Software engineering economics. *Software Engineering, IEEE Transactions on SE-10*, 1 (Jan. 1984), 4 –21.

[3] Briand, L., Daly, J., and Wust, J. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on 25*, 1 (Feb. 1999), 91 –121.

[4] Card, D. N., Page, G. T., and McGarry, F. E. Criteria for software modularization. In *Proceedings of the 8th international conference on Software engineering* (Los Alamitos, CA, USA, 1985), ICSE '85, IEEE Computer Society Press, p. 372377.

[5] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. 20*, 6 (June 1994), 476493.

[6] Eden, A. H., and Mens, T. *Measuring Software Flexibility.*

[7] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] Fowler, M. J., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*, 1 ed. Addison-Wesley Professional, July 1999.

[9] Halstead, M. H. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, USA, 1977.

[10] Hitz, M., and Montazeri, B. Measuring coupling and cohesion in object-oriented systems. In *Proc. Intl. Sym. on Applied Corporate Computing* (1995).

[11] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, p. 485495.

[12] Liskov, B. Keynote address - data abstraction and hierarchy. *ACM SIGPLAN Notices 23*, 5 (May 1988), 17–34.

[13] Martin, R. C. Design principles and design patterns.

[14] Martin, R. C. The dependency inversion principle. *C++ Report* (1996).

[15] McConnell, S. *Code complete.* Microsoft Press, Redmond, Wash., 2004.

[16] Oman, P., and Hagemeister, J. Construction and testing of polynomials predicting software maintainability. *J. Syst. Softw. 24*, 3 (Mar. 1994), 251266.

[17] Parnas, D. On the design and development of program families. *IEEE Transactions on Software Engineering SE-2*, 1 (Mar. 1976), 1–9.

[18] Ryder, B. G., and Tip, F. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), PASTE '01, ACM, p. 4653.

[19] Selby, R., and Basili, V. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering 17*, 2 (Feb. 1991), 141–152.

[20] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. N degrees of separation. ACM Press, pp. 107–119.

[21] Von Mayrhauser, A., and Vans, A. Program comprehension during software maintenance and evolution. *Computer 28*, 8 (Aug. 1995), 44–55.

Appendix A

Appendix A, Metrics

For data on the lines of code, see 5.1.
Because of the size and format of the metrics these are contained in a separate spreadsheet.

Appendix B

Appendix B, Bootstrapping code

```
public static class ApplicationBootstrapper
{
 //All assemblies of the application
 private static readonly string[] _neededAssemblies = new []
 {
  "NHibernate",
  "NHibernate.Abstractions",
  "NHibernate.Cache",
  "NHibernate.DomainModel",
  "NHibernate.Event",
  "NHibernate.Hql",
  "NHibernate.Linq",
  "NHibernate.Sql",
  "NHibernate.Types",
  "NHibernate.Util"
 };

 /// <summary>
 /// Static setup method for initializing the application
 /// </summary>
 public static void SetUp()
 {
  var assembliesInAppDomain = AppDomain.CurrentDomain
```

```
    .GetAssemblies();

  //Ensure all assemblies are loaded, if not load them
  _neededAssemblies.Where(x => !assembliesInAppDomain
    .Any(y => y.FullName == x))
    .ToList()
    .ForEach(x => Assembly.Load(x))
    ;

  //Iterate over all nhibernate assemblies
  foreach (var nhAssembly in AppDomain.CurrentDomain
    .GetAssemblies()
    .Where(e => e.FullName.StartsWith("NHibernate")))
  {
    BootstrapAssembly(nhAssembly);
  }
}

/// <summary>
/// Bootstrap a single assembly
/// </summary>
/// <param name="nhAssembly">The assembly to bootstrap</param>
private static void BootstrapAssembly(Assembly nhAssembly)
{
  //Find the first type implementing the IAssmeblyBootstrapper
  //interface
  var bootstrapType = nhAssembly.GetTypes()
    .FirstOrDefault(
      typeof(IAssemblyBootstrapper).IsAssignableFrom
    );

  if (bootstrapType != null &&
    bootstrapType.GetConstructors().Any())
  {
  //use reflection to instantiate the type
    var setupInstance = bootstrapType.GetConstructors()[0]
      .Invoke(new object[0]) as IAssemblyBootstrapper;

    //call the setup method
    setupInstance.Setup();
    }
  }
}

/// <summary>
/// Interface for a class that contains setup code for an assembly
/// </summary>
public interface IAssemblyBootstrapper
{
  void Setup();
}
```

Appendix C

Appendix C, Metric collection code

Below is the code for the RFC walker and the coupling walker. Concurrent dictionaries are used as the calling code executes these walkers in parallel to speed up the collection of metrics. Because these are concurrent the addition of elements to these dictionaries might seem a bit awkward.

```
public class RfcWalker : SyntaxWalker
{
 public readonly ConcurrentDictionary<string, int>
  RfcByFullname =
  new ConcurrentDictionary<string, int>();

 private string _rootPath;

 public RfcWalker(string rootPath)
 {
  _rootPath = rootPath ?? Environment.CurrentDirectory;
 }

 /// <summary>
 /// Visit the interface declarations
 /// </summary>
 public override void VisitInterfaceDeclaration
  (InterfaceDeclarationSyntax node)
```

```
{
 base.VisitInterfaceDeclaration(node);
 GetRfcValuesForType(node);
}

/// <summary>
/// Visit the class declarations
/// </summary>
public override void VisitClassDeclaration
 (ClassDeclarationSyntax node)
{
 base.VisitClassDeclaration(node);
 GetRfcValuesForType(node);
}

/// <summary>
/// Calulate the values for RFC for a type
/// </summary>
private void GetRfcValuesForType(TypeDeclarationSyntax node)
{
 //Create a compilation, we need this to determine the typenames of varia
 var compilation =
 Compilation.Create("TestCompilation")
  .AddReferences(
   new AssemblyNameReference("mscorlib"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.sql.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.hql.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.util.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.linq.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.event.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug\nhibernate.abstractions.dll"),
   new AssemblyFileReference(_rootPath +
     @"\Nhibernate\bin\debug\nhibernate.types.dll"))
    .AddSyntaxTrees(node.SyntaxTree);

  //Get the semantic model telling something about the types
  //of the class
  var model = compilation.GetSemanticModel(node.SyntaxTree);

  //get the symbol of the current class
  var classSymbol = model.GetDeclaredSymbol(node);

  //Get all the public methods and properties of the class
  var methods = node.Members
   .OfType<MethodDeclarationSyntax>()
   .Count(x =>
    x.Modifiers.Any(mod =>
     mod.Kind == SyntaxKind.PublicKeyword
    ) || node is InterfaceDeclarationSyntax);
  var properties = node.Members
   .OfType<PropertyDeclarationSyntax>()
   .Count(x =>
    x.Modifiers.Any(mod =>
     mod.Kind == SyntaxKind.PublicKeyword
    ) || node is InterfaceDeclarationSyntax);

  //Add them to the dictionary
  RfcByFullname.AddOrUpdate(
   classSymbol.ToDisplayString(),
   methods + properties,
   (key, value) => methods + properties);
  }
}


public class ImportCouplingWalker : SyntaxWalker
{
 private string _rootPath;

 public ImportCouplingWalker(string _rootPath)
 {
  this._rootPath = _rootPath ?? Environment.CurrentDirectory;
```

```
}

public readonly ConcurrentDictionary<string, int>
 ImportClassCouplingByFullname =
 new ConcurrentDictionary<string, int>();
public readonly ConcurrentDictionary<string, int>
 ImportInterfaceCouplingByFullname =
 new ConcurrentDictionary<string, int>();
public readonly ConcurrentDictionary<string, IEnumerable<string>>
 CoupledTypesByFullname =
 new ConcurrentDictionary<string, IEnumerable<string>>();

public override void VisitClassDeclaration
 (ClassDeclarationSyntax node)
{
 GetCouplingValuesForType(node);
}

public override void VisitInterfaceDeclaration
 (InterfaceDeclarationSyntax node)
{
 GetCouplingValuesForType(node);
}

public void GetCouplingValuesForType
 (TypeDeclarationSyntax node)
{
 //Create a compilation, we need this to determine the
 //typenames of variables.
 var compilation = Compilation.Create("TestCompilation")
  .AddReferences(
   new AssemblyNameReference("mscorlib"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug-2.0\nhibernate.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug-2.0\nhibernate.sql.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug-2.0\nhibernate.hql.dll"),
   new AssemblyFileReference(_rootPath +
    @"\Nhibernate\bin\debug-2.0\nhibernate.util.dll"),
    new AssemblyFileReference(_rootPath +
     @"\Nhibernate\bin\debug-2.0\nhibernate.linq.dll"),
    new AssemblyFileReference(_rootPath +
     @"\Nhibernate\bin\debug-2.0\nhibernate.event.dll"),
    new AssemblyFileReference(_rootPath +
     @"\Nhibernate\bin\debug-2.0\nhibernate.abstractions.dll"),
    new AssemblyFileReference(_rootPath +
     @"\Nhibernate\bin\debug-2.0\nhibernate.types.dll"))
   .AddSyntaxTrees(node.SyntaxTree);

//Get the semantic model telling something about the types
//of the class
var model = compilation.GetSemanticModel(node.SyntaxTree);

//Find the types of properties in the class
var propertyTypes = node.Members
 .OfType<PropertyDeclarationSyntax>().Select(x => x.Type);
//Find the types of fields in the class
var fieldTypes = node.Members
 .OfType<FieldDeclarationSyntax>()
 .Select(x => x.Declaration.Type);
//Find the types of all paramters of methods
var parameterTypes = node.Members
 .OfType<MethodDeclarationSyntax>()
 .SelectMany(x => x.ParameterList.Parameters)
 .Select(param => param.Type);
//Find all types of parameters in the constructor
var constructorTypes = node.Members
 .OfType<ConstructorDeclarationSyntax>()
 .SelectMany(x => x.ParameterList.Parameters)
 .Select(param => param.Type);
//Find the types of all local variables
var localVariableTypes = node.Members
 .OfType<MethodDeclarationSyntax>()
 .Where(x => x.Body != null)
 .SelectMany(x =>
  x.Body.DescendantNodes()
    .OfType<LocalDeclarationStatementSyntax>()
 ).Select(var => var.Declaration.Type);
```

```
//Concatenate the 5 lists
var allTypes = propertyTypes
     .Union(fieldTypes)
     .Union(parameterTypes)
     .Union(constructorTypes)
     .Union(localVariableTypes);

//Get the types of arrays these are handled differently
allTypes = allTypes
 .Union(allTypes.OfType<ArrayTypeSyntax>()
  .Select(x => x.ElementType)
 );
//Get the types that use a qualified name (e.g. Dialect.Dialect)
allTypes = allTypes
 .Union(allTypes.OfType<QualifiedNameSyntax>()
  .Select(x => x.Right)
 );

//Exclude the predefined types (these are types like int etc.)
var stableTypes = allTypes.Where(x => x is PredefinedTypeSyntax);

//get all the non stable types
var nonStableTypes = allTypes.Except(stableTypes);

//Get a list of all symbols in the class
var symbols = GetAllSymbolsRecursive(
 nonStableTypes.Select(x => model.GetTypeInfo(x).Type), model
);

//get the symbol of the current class
var classSymbol = model.GetDeclaredSymbol(node);

//Count all elements, that start with NHibernate, not being the
//current class and an predicate for classes of interfaces
var importedClasses = symbols
 .Where(x => x.ToDisplayString().StartsWith("NHibernate") &&
  x.ToDisplayString() != classSymbol.ToDisplayString() &&
  x.TypeKind == TypeKind.Class).ToArray();
var countImportedClasses = importedClasses.Count();
var importedInterfaces = symbols
```

```
  .Where(x => x.ToDisplayString().StartsWith("NHibernate") &&
   x.ToDisplayString() != classSymbol.ToDisplayString() &&
   x.TypeKind == TypeKind.Interface).ToArray();
 var countImportedInterfaces = importedInterfaces.Count();

 //Make sure we get distinct types
 var referencedTypes = importedClasses.Union(importedInterfaces)
  .Select(x => x.ToDisplayString()).Distinct().ToArray();

 //get the fullname of the current class
 var typeFullName = classSymbol.ToDisplayString();

 //Add the results to the proper dictionary
 CoupledTypesByFullname.AddOrUpdate(typeFullName,
  referencedTypes,
  (key, value) => referencedTypes);
 ImportClassCouplingByFullname.AddOrUpdate(typeFullName,
  countImportedClasses,
  (key, value) => countImportedInterfaces);
 ImportInterfaceCouplingByFullname.AddOrUpdate(typeFullName,
  countImportedInterfaces,
  (key, value) => countImportedClasses);
}

/// <summary>
/// Find all symbols recursively
/// </summary>
/// <param name="symbols">Existing list of symbols</param>
/// <param name="model">Model to get the symbols from</param>
/// <returns></returns>
private IEnumerable<TypeSymbol> GetAllSymbolsRecursive
 (IEnumerable<TypeSymbol> symbols, SemanticModel model)
{
 var namedTypeSymbols = symbols.OfType<NamedTypeSymbol>();

 var genericSymbols = namedTypeSymbols.Where(x => x.IsGenericType)
  .SelectMany(x => GetAllSymbolsRecursive(
   x.TypeArguments.ToList(), model)
  );
```

```
    //concatenate the newly found list with the old one
    symbols = symbols.Union(genericSymbols);

    return symbols;
  }
}
```