

**“Assessing the impact of API evolution”**

*An experiment with C++ and the Qt Framework*

by

Taco Christiaan Witte

in Partial Fulfillment of the Requirements for the Degree of  
**Master of Software Engineering**

at the University of Amsterdam

August 6, 2010

Supervisors:

Dr. Tijs van der Storm, CWI and University of Amsterdam

Dr. Jan C. de Munck, VU University Medical Center

# Abstract

Practically all programs use one or more libraries. Such third-party libraries are subject to software evolution like all other software. New versions of libraries are released and those versions will sometimes not be backwards compatible. Migrating a program to a newer version of a library can be hard. It would be useful to know how hard a specific migration is. Currently, there is no method to get such information.

How can the impact of API evolution on a program be measured automatically? This study proposes a technique called Library Changes Impact Analyzer (LICIA). It consists of two steps to answer this question. First, the API changes are manually looked up and written down in a certain format. Second, the program is automatically analyzed to find references to parts of the API that have been changed. The result is a spreadsheet that shows which source code lines in the program should be adapted and which changes have an effect on them.

The experiment done to assess the feasibility of the solution is the migration of a brain image analysis program written in C++ to a newer version of the Qt Framework. This experiment has been chosen because the program and the framework have a size and complexity that is comparable to those found in real migrations and because enough information is available to conduct the experiment.

The migration is successful. Precision is 81% and recall is 96%. 19% of the results is superfluous. The false negatives are mostly caused by API changes that are not mentioned in the documentation. The false positives are mostly caused by one mistake in the survey and two assumptions in LICIA. The superfluous results are caused by not taking into account the specific change type in the impact analysis.

LICIA doesn't produce an overview of all code locations on which API changes have an effect, as stated in the hypothesis, because recall is not 100%.

The main contribution of this work is a technique to get a precise overview of the effect of API changes on a program. Another contribution is a list of refactorings that were found in the experiment.

## Keywords:

software evolution, software maintenance, API evolution, framework evolution, refactoring, framework, library, component reuse, code reuse, backwards compatibility, partial program analysis, C++, abstract syntax tree

## Primary classification:

D.2.7 Software/Software Engineering/Distribution, Maintenance, and Enhancement

## Preface

This thesis is the result of a master's project carried out from March 19 to July 30, 2010.

I would like to thank my supervisors Tijs van der Storm and Jan de Munck for giving me an interesting and good research project and the freedom to conduct it in my way. Especially Tijs helped me to start well and construct a good research plan. Both have given me useful feedback and support along the way.

Also I would like to thank my teachers Hans Dekkers and Paul Griffioen for their practical approach to computer science teaching and the many things I learned from them. This master was a very good fit for me and they are the main people who created it.

Most of all I would like to thank my parents for their support during this project and for giving me the opportunity to focus completely on the project.

Taco Witte  
Amsterdam, The Netherlands  
July 2010

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Problem . . . . .	1
1.2. Hypothesis . . . . .	2
1.3. Experiment . . . . .	2
1.4. Scope . . . . .	3
1.5. Structure . . . . .	3
<b>2. Context</b>	<b>4</b>
2.1. Theory . . . . .	4
2.2. Approaches to API migration in practice . . . . .	6
2.3. Tool-supported migration . . . . .	6
<b>3. Research method</b>	<b>8</b>
3.1. Technique . . . . .	8
3.2. Alternative choices . . . . .	9
3.3. Why this technique . . . . .	10
3.4. Experiment . . . . .	10
3.5. Why this experiment . . . . .	11
3.6. Validation and analysis . . . . .	11
<b>4. Experiment</b>	<b>13</b>
4.1. Brain Image Analysis Package . . . . .	13
4.2. Survey . . . . .	13
4.3. Impact analysis with LICIA . . . . .	16
4.4. Results . . . . .	18
4.5. Replicating the experiment . . . . .	20
<b>5. Analysis</b>	<b>21</b>
5.1. Migrating the program . . . . .	21
5.2. Threats to validity . . . . .	24
5.3. Implications for the techniques . . . . .	24
5.4. Improvements . . . . .	25
<b>6. Conclusion</b>	<b>26</b>
6.1. Implications for other migrations . . . . .	26
6.2. Future work . . . . .	27
<b>Glossary</b>	<b>29</b>
<b>Bibliography</b>	<b>31</b>

## Contents

<b>A. Migration advice</b>	<b>33</b>
A.1. Strategy . . . . .	33
A.2. Survey of API changes . . . . .	34
A.3. Impact analysis . . . . .	34
A.4. User Interface files . . . . .	35
A.5. Analysis . . . . .	36
A.6. Advice . . . . .	37
<b>B. Change categories</b>	<b>39</b>

# Chapter 1.

## Introduction

This chapter first introduces the problems encountered by migrating a big program to a newer version of an API. The problems are analyzed and a solution is proposed. This study investigates the feasibility of the solution with an experiment.

### 1.1. Problem

Practically all programs use one or more libraries<sup>1</sup> developed by a third party. This reduces the work to develop an application. The intention of using libraries is to isolate code that can be reused among a group of programs and make it available with an abstract interface. This is the principle of modularity. The abstract interface shields programs that use it from changes within the library.

Such third-party libraries are subject to software evolution like all other software. The functionality of the library will change, design flaws will be found and the abstract interface will have to be adapted accordingly.

New versions of libraries are released and those versions will sometimes not be backwards compatible. If there is a reason to switch the program to such a newer version, the source code of the program will have to be migrated to it. At least four reasons can exist for a migration. First, the new version contains new functionality that the program can use. Second, problems in the old version are fixed in the new one. Third, using the new version makes distribution of the program easy because the new version is already installed on computers or because the manufacturer of the program doesn't have to worry about the distribution of the library. Fourth, the new version supports new computers that the old version doesn't support.

Migrating a program can be hard. It can be hard to understand how the migration should be done. How hard depends on how much impact the changes have on the specific program: how many changes there are, the kind of changes and especially how much changed functionality is used by the program.

It would be useful to know how hard a specific migration is and what makes it hard. This would allow programmers to plan the migration in an appropriate way. For example,

---

<sup>1</sup>The concepts “API”, “library” and “framework” are used interchangeably in this document. More precisely, an Application Programming Interface (API) is the interface with which the functionality of a library or framework is exposed to programs that use it.

programmers could decide to migrate a small part first as a test case, or programmers could decide not to migrate at all because the costs are too high.

Currently, there is no method to measure how hard a migration is. It's not feasible to measure this manually, because such an analysis is most useful when there are many changes.

This leads to the following research question: *How can the impact of API evolution on a program be measured automatically?*

## 1.2. Hypothesis

API evolution has impact on a program if the program should be adapted to let it work as expected with the new version. If the program works as expected with the new version of the API without being modified, the changes in the API have no impact. Whether a program works as expected can be determined by specifying the expected output for a given input and verifying that the program works according to those specifications.

Therefore, the number of code locations in the program on which API evolution has impact is a measure for the impact of API evolution on a program. This is an upper boundary for the code locations that need to be adapted because some code locations on which changes have an effect don't need to be adapted. For example, suppose that the new API declares the class *ColorWidget* in the header file *colorwidget.h* while the old version declared it in *widgets.h*. This change has an effect on each use of *ColorWidget* because the class cannot be found now. However, including *colorwidget.h* solves the problem for all uses of *ColorWidget* in a source file. If *ColorWidget* is used 5 times in that source file, the effect of the change on 5 code locations is solved by adapting 1 code location.

This report studies LIbrary Changes Impact Analyzer ([LICIA](#)), a technique that finds the code locations on which API evolution has impact. The hypothesis is that LICIA produces an overview of all code locations on which API changes have an effect.

## 1.3. Experiment

The hypothesis cannot be accepted at face value. For example, LICIA may only find a subset of changes and would therefore not be usable as a basis for decisions.

Therefore, LICIA is tested with an experiment. A big program is migrated to a newer API. The department Physics and Medical Technology at the VUmc university hospital develops and uses a program to study medical images. The program was developed over a period of 25 years and is still in use. It can be used to analyze, segment and visualize images from CT, MRI, PET, MEG and other measurements. Its name is Brain Image Analysis Package ([BIAP\[dM10\]](#)).

BIAP uses an old version of the Qt framework. It uses version 2.3.2. That version is not maintained anymore and will soon stop working on new computers.

The current version of Qt (4.x) has functionality that they would like to use such as

support for OpenGL. The wish is to migrate BIAP from Qt 2.3.2 to Qt 4.

Qt is a framework for C++ developed by Trolltech/Nokia. It's mainly used to create graphical user interfaces but also offers much lower level functionality such as an event system. It's platform-independent. The framework is provided with extensive documentation and source code. It's widely used so there's a lot of information available about it on-line.

There is documentation that describes the changes, but their effect on the program is not clear. Some changes are big, such as those in Qt Designer which is used to design the dialog windows.

Adapting the program to a newer version of Qt was attempted in the past and seemed too much work to do without strategy.

## 1.4. Scope

The second API is a newer version of the first API. This implies that the design is the same and the APIs are identical apart from a set of visible differences. The program only needs to be adapted to those differences. A visible difference is a difference that is described in the documentation and that can be seen in the public interface.

Migrating to essentially unrelated APIs is not in the scope of this study.

## 1.5. Structure

Related approaches and theory are described in chapter [Context](#).

Chapter [Research method](#) describes the technique, how it will be applied and how it will be evaluated.

The experiment and its results are described in chapter [Experiment](#).

The implications of the experiment are described in chapter [Analysis](#).

The [Conclusion](#) summarizes this study, describes the implications for other migrations and provides starting points for future research.

A glossary is provided that clarifies the use of some jargon in this report.

The appendices provide extra information. Specifically, [Migration advice](#) uses the knowledge obtained in this study to give advice for the next migration step.

# Chapter 2.

## Context

This chapter describes API migrations, approaches to API migration in practice and related work.

### 2.1. Theory

Before describing how API migrations are done in practice, it's useful to consider what APIs are, why they change, how those changes can be classified and how the change process works.

#### 2.1.1. API migration

Application Programming Interfaces (APIs) form the interface with which functionality of a library or framework is exposed to programs. They offer functionality that is useful for many programs.

All non-trivial programs use libraries and frameworks<sup>1</sup> in some way to reuse code and reduce work.

APIs expose functionality so programs can use it. They expose it with abstract and stable interfaces so programs don't need to be adapted for every little change made in a library or framework.

Libraries evolve just like other software. [LR01] describes how demands change and software has to adapt to stay relevant. Certain library changes cannot be made without changing the API.

Programs have to be adapted when an API they use changes. This process is called [API migration](#).

#### 2.1.2. Refactorings and other changes

To develop a new version of an API, programmers make many small changes to add functionality, to remove functionality or to fix existing functionality. Seeing the difference

---

<sup>1</sup>[TDX07] describes the difference between a library and a framework: in frameworks, methods call each other while in libraries methods are only called by the program that uses the library.

between two versions as a set of small changes makes it easier to understand what changes.

The small changes are refactorings or non-refactorings. A [refactoring](#) is a “program transformation that changes the structure of a program but not its behavior” [DJ06]. For example, if a function is moved to another class, the structure of the program changes but its behavior doesn’t. Moving a function is a refactoring. Changes are non-refactorings if they change the behavior of the program. For example, if a sorting algorithm is improved to take into account the input language, this changes the behavior of the program. This is a non-refactoring.

About 80% of changes are refactorings [DJ06]. Because refactorings only change the structure of code, it’s possible to automatically adapt other code to them. This is the theoretical basis for (semi-)automatic API migration.

Several types of refactorings are described in the literature. The most complete list seems to be in [FBB<sup>+</sup>99] and its companion website [Fow10]. The list in the appendix in chapter [Change categories](#) uses names mentioned in the book and on the website if they are available.

### 2.1.3. Change process

An important aspect of API migration is the process used to introduce new functionality and remove old functionality.

If functionality is added there is no problem, because this is a backwards-compatible change. The problem is when functionality is changed or removed, because those changes are in general backwards-incompatible. Programs that use the library expect that certain functionality is present under a certain name and have to be adapted if that is no longer true.

[Obsolete functionality](#) is functionality that has already been replaced by new functionality but is temporarily kept in code. This makes the change process more graceful for programmers. Migration can be done in steps. Code that uses the changed functionality does not break but programmers are strongly suggested to update. This process is called the [deprecate-replace-remove cycle](#) in the literature.

The deprecate-replace-remove cycle makes it harder to identify changes [TDX07] [DJ06]. The reason is that the change is not made in one version. The old functionality is removed several versions after the new functionality was introduced. For instance, old functionality is removed in the next major version in Qt<sup>2</sup>.

In this study it will be assumed that obsolete functionality cannot be used. This makes the analysis of changes between versions clearer by mentioning changes where they occur. Also, obsolete functionality will eventually be removed. Using obsolete functionality only postpones the burden of change.

---

<sup>2</sup>The porting documentation for Qt 3 and 4 states “Qt X includes many additional features and discards obsolete functionality.”.

## 2.2. Approaches to API migration in practice

### 2.2.1. Manually migrating

A common approach to API migration is to manually adapt a program to the new API without using any special tools. This can be done in small steps when new versions of the API are published, or in big steps. For instance, two versions of a program can be maintained. One is the development version; it contains the newest functionality and follows changes in the API. The other is the production version; it's only modified to correct errors and does not follow changes in the API. Once in a while, a new production version is created from the development version.

This is a straightforward approach. No new tools need to be learned and all work appears directly in the source code. However, during the migration the program will not compile because the program uses partly the old API and partly the new one. This means that modifications can only be verified when the migration is finished.

### 2.2.2. Not migrating at all

Another approach is to avoid the problem, either by not migrating the program or by writing it from scratch using the new API. If the program is not migrated, it will stop being useful at some stage. The old API stops working on newer operating systems or cannot be used anymore for some other reason.

Starting from scratch with the new API can sometimes be a good choice. Technologies and wishes of end users may have changed enough to merit a fresh look. This is however probably the most expensive choice.

## 2.3. Tool-supported migration

A migration has several aspects. This includes the program that is migrated, the library that changes and other programs that may have been migrated before. All aspects offer opportunities to solving the problem of migrating a certain program to a certain newer library.

[CN96] finds a solution in the change process of the library. The idea is that the manufacturer of the library adds annotations to it. Those annotations describe what was changed and how a program should be adapted to that change. This makes the migration to a newer version more or less automatic. If a library contains such annotations this is an option. Qt doesn't have them so this is not viable.

[ZTX<sup>+</sup>10] finds a solution in other programs that were confronted with the same problem. It proposes to find several programs that use the same API and have already been migrated to the new version. By mining the changes in those programs, changes for this program could be deduced. This idea is only proposed and not studied in that paper. The paper studies migration to a different programming language while using a certain API that is available for both languages. If the idea would work for migrating to a newer

API in the same programming language, it could be a viable option for this program. Qt is widely used and many programs that use it are free software, so there would be code available for mining.

SemDiff[DR08] finds a solution in how the framework itself was adapted. This only works if methods in the framework call each other. It mines the revisions of the framework and looks for source code differences. It uses the information to give smart ad-hoc suggestions to adapt the program. The program is configured to use the new API and the suggestions aim to solve compilation errors. The total number of code modifications is only known at the end. If the full revision history is available this solution is an option. In the case of Qt most of the source code is available, but the revision history is not. This is not a viable solution here.

Diff-CatchUp[XS07] and AURA[WGAK10] are similar to SemDiff[DR08] but they don't use the revision history of the library. Diff-CatchUp[XS07] only uses the public API description. AURA[WGAK10] uses the source code of the two versions of the library and recognizes more complex changes. Both work by finding elements in the new API that correspond to elements in the old API and suggesting to use those new elements. The kind of changes that they can find is limited. These techniques cannot help in estimating how many modifications will be required. This means that they don't solve the problem addressed in this study.

Library Changes Impact Analyzer (LICIA), the technique proposed in this study, finds the solution in the preparation of the migration. If the difficulties in the migration are known in advance, this makes the migration easier because the migration strategy can be tailored to the specific migration. It may also result in no migration if the migration turns out to be too big. If the documentation is used to find API changes (which is the case in the experiment), it should be available and of high quality. This appears to be the case with Qt. If the documentation is not used, an automatic method such as AURA[WGAK10] is needed to analyze the API differences.

The idea of categorizing API changes is based on [BCLvdS10]. That study focuses on migrating to essentially different APIs in contrast to this study.

# Chapter 3.

## Research method

This chapter describes the techniques in LICIA, how feasibility will be evaluated with an experiment and how the results will be analyzed.

### 3.1. Technique

LICIA is a technique for finding how changes in an API have an effect on a (big) program that uses it.

It produces a list of code locations that should be adapted in a program, together with information about what changes have an effect on those locations. These results provide information about which source files are most affected by changes and which API classes cause most of the changes.

The technique consists of a survey of API changes and an impact analysis.

#### 3.1.1. Survey of API changes

The result is a change descriptions spreadsheet. It contains the classes, functions and other identifiers in the API that are changed from one version of the API to the next. Those changes are classified as a certain type of change (e.g. rename function). Another result is a list of change types and their level of adaptation. It indicates how hard it's expected to be to adapt to such a change type.

Which classes, functions and other identifiers are changed is derived from the documentation provided by the manufacturer of the API. Such documentation describes which parts of the API are changed and how to port a program that uses the old version to the new version. Whenever a class, function or other identifier is mentioned, it's added to the list.

The list of change types is obtained by analyzing the list of changed classes, functions and other identifiers. For each item, the intention is to derive from the documentation and/or public API how it was changed. For instance, a function may be renamed. Common names are used if they are available. Otherwise new names are invented.

The levels of adaptation that are proposed in this study are (0) *compatible change*, (1) *cosmetic*, (2) *composition*, (3) *contract change* and (4) *missing*. A higher [level of adaptation](#) implies a higher difficulty to adapt a program to a change. Change types are

categorized into these levels of adaptation. This is a manual process. The way in which these levels of adaptation are interpreted is described in chapter [Experiment](#).

This classification of changes into change types and change types into levels of adaptation helps in getting an idea about how hard it will be to adapt the program to the change. Numerous simple changes can be distinguished from a few changes that are hard to adapt to.

### 3.1.2. Impact analysis

The result of the impact analysis is a spreadsheet listing all code locations in the program on which API changes have an effect.

The impact analysis uses the survey from the previous step, the source code of the program and the old version of the API. It consists of three steps.

First, the source code of the program is parsed to extract all references to the API. References are classes, functions or other identifiers of the API that are mentioned in the source code. As an intermediate step, a parser is used that constructs an [abstract syntax tree \(AST\)](#) in which types are resolved. Otherwise, references would be missed or not found correctly.

Next, inheritance is taken into account. The inheritance hierarchy for the given source code is used to determine which classes are subclasses of a given class. We assume that subclasses of a changed class are changed in the same way. We expect that this works because the generally adopted Liskov substitution principle states that subclasses should obey their superclass's contract.

Finally, the impact analysis algorithm is used to determine which references are related to functionality that is changed. This is the actual impact analysis. If the program uses functionality that has been changed, the change has an effect on the program. Those API items that are not referenced are ignored. It depends on the type of change whether the change has an effect on uses of only one function of a class or all uses of a class.

## 3.2. Alternative choices

The information about API changes is derived from the documentation by hand. It can also be derived from the public API or from the source code of the library with a tool. It's hard to find changes with a tool[DJ06]: "it is hard to discover the changes in a large system". That's why this study uses the documentation.

The impact analysis is done automatically. This is necessary because doing the analysis manually is too much work: all of the program's source code should be processed several times to find references to changed classes, functions and other identifiers.

An alternative to using an abstract syntax tree to extract references to the API is to simply do a textual analysis. In that case, a textual analysis tool such as *grep* would be used to find all occurrences of classes, functions or other identifiers in the source code. That is easier to implement but much less precise because it doesn't take into account

the data types. For example, if the class *QCursor* is changed and some variable in the program has the same name the two are not related. A textual analysis would find that variable. Also, many references would not be found because they don't contain the complete name of the functionality that has been changed. This is for example the case when a value is returned from a function; the data type of the value is not necessarily explicitly mentioned in the code that calls the function.

### 3.3. Why this technique

It's likely that LICIA yields results for four reasons.

First, measuring the effect of API changes by checking all references to the API in the source code of the program is sufficient. The program cannot rely on API functionality if it doesn't refer to it. This implies that the impact analysis is in theory complete.

Second, the API classes, functions and identifiers will be identified correctly because the references to them are unambiguous and unique. The reason is that a parser is used to resolve data types in the source code. This implies that the code locations in the program that will be found really are linked to the data types in the API and don't just have names that are similar by coincidence. It also implies that references will be found even if the data types of the API are not mentioned explicitly. This is the case when a data type is not constructed in the program but returned by a library function.

Third, changes will also be found if they are inherited from another class. All classes that inherit from a changed API class and are known in a source file are searched in the analysis.

Fourth, dynamic binding is handled correctly. If a function is a [virtual function](#) it's impossible to tell by static analysis which function will be called at run-time. This analysis uses static code analysis. However, it's not necessary to know which function will be called at run-time as long as the class in which the virtual function is declared is checked for changes. Classes that inherit from the class in which the function is declared should respect the contract of their superclass.

### 3.4. Experiment

The techniques described in this chapter are evaluated in an experiment. The experiment is the migration of BIAP, a brain image analysis program developed at VUmc from Qt Framework version 2.3.2 to 4.6.

The migration from Qt 2.3.2 to Qt 4 is split into two migrations, first from Qt 2.3.2 to Qt 3 and then from Qt 3 to Qt 4. This is logical because each [minor version](#) is backwards-compatible and the porting documentation describes how to upgrade from the previous [major version](#). By upgrading one major version at a time, the documentation can be used.

LICIA is applied in the first migration, from Qt 2.3.2 to Qt 3. The results are used to evaluate the techniques.

Then LICIA is applied in the second migration, from Qt 3 to Qt 4. The quality of the impact analysis can be estimated with the knowledge of the previous migration. This results in a migration advice. Doing and evaluating the second migration is not part of this study.

## 3.5. Why this experiment

A good experiment for testing these techniques should meet two conditions.

First, the program and framework should have a size and complexity comparable to those found in real migrations. An artificial experiment with a small program or a small framework may hide practical problems that would not have been expected in advance and that make the techniques unfeasible in practice.

Second, enough information should be available to conduct the experiment. The source code of the program should be available. The framework should be well documented, especially as far as migrations are concerned.

Both conditions are met in this experiment.

The program is not a simplified case but a real program. It consists of 226,000 lines of C++ and C code. 64,000 of those lines are in modules that use Qt. BIAP has 50+ dialogs with tabs and tens of buttons. The Qt framework is also real.

The source code of BIAP is available and the author cooperates in this study. The Qt framework is provided with extensive documentation and (partly) source code. The documentation explicitly describes what changes from one version to the next.

## 3.6. Validation and analysis

The technique is validated by adapting the program to the changes found in the impact analysis. For example, if the list mentions that a renaming of a class has an effect on a certain line of code, the name of the class is changed on that line.

### 3.6.1. Success of migration

The source code obtained after making those adaptations is compiled with the new version of the API. If the resulting program works like the original program with the original API, except for acceptable small differences, the migration is successful. No test suite is available in the experiment so the comparison will be done manually.

### 3.6.2. Quality

False positives and negatives are studied to determine the quality of the impact analysis if the migration is successful. Those code locations that did not need adaptations are false positives. Those code locations that were not in the list but needed adaptations

to migrate the program are false negatives. Chapter [Experiment](#) describes how this is determined.

A low number of false negatives is much more important than a low number of false positives, because it means that programmers are not notified about some code and have to look for it themselves. That takes more effort than determining that a specific change has no effect on a certain line of code.

# Chapter 4.

## Experiment

This chapter describes how LICIA is applied in the experiment and what results are obtained. The survey is performed on the API changes in Qt. The impact analysis is performed on a brain image analysis program.

### 4.1. Brain Image Analysis Package

Brain Image Analysis Package (BIAP) consists of several parts. There are 4 parts that interact with Qt.

One is a library with user interface elements. It's called *Q2\_ui*. It's a mixture of C++ code, header files and Qt UI files. The Qt UI files are used to generate C++ source code and header files. The generated files are used in the analysis because the Qt UI files cannot be parsed directly.

The other three are programs that use those user interface elements and the other parts of the program to do their work. They are called QView2D, QMatchVol and QDataEditor. QView2D is a relatively simple application and can be used as a test case.

### 4.2. Survey

The survey studies changes from Qt version 2.3.2 (the version used by BIAP) to 3.0.1 (as an intermediate step towards 4). If the program works with version 3.0.1 it will also work with the most recent version in the 3.x series.

Qt includes extensive documentation about the migration to Qt 3[Tro02]. This documentation describes all changes that are not backwards compatible. It also describes how to adapt a program to those changes.

All changes mentioned in the documentation are categorized as a certain type of refactoring or non-refactoring and added to a list. Refactorings are categorized with commonly used names if they are available. If not, a new name is invented. All those refactorings and non-refactorings are categorized as a certain level of adaptation.

If functionality is removed, this is considered to be a non-refactoring. Removing functionality means that other functionality should be used. In general, that other functionality doesn't have exactly the same behavior and a similar structure.

Changes that are backwards compatible (level of adaptation 0) are ignored because they have no effect on a program. For example, if functionality is added this is a backwards compatible change.

Table 4.1 gives an example of a change type for each level of adaptation. They are described now.

Changes are of level *cosmetic* if the same functionality is present in the new version and only a simple small change is required to use it. For example, *rename function* is such a change. All references to a certain function should be adapted to the new name of the function. Each such adaptation is simple and small because only one word has to be replaced with another word.

Changes are of level *composition* if the same functionality is present but some extra calls are required to use it, possibly in addition to characteristics of level *cosmetic*. There are no changes in the contract of functions, so there is no need for pre or post processing of data. For example, *split function* is such a change. The functionality of one function is divided over several functions. References to the old function should be replaced with a list of calls to the new functions. No code is required except for function calls to the new functions, so this change is not of type *contract change*.

Changes are of level *contract change* if the same functionality is present and there is a change in contract, possibly in addition to characteristics of level *composition*. This means that code should be adapted by calling one or more functions and adding code for pre and/or post processing of data. For example, *change post condition* is such a change. A function has a different return type or different guarantees about the value returned by the function. Code on which this change has effect should be adapted to the difference by for example converting the return value to the old type.

Changes are of level *missing* if they don't fit the requirements of another level. For example, *remove class* is such a change. A complete class is removed from the API and there is no equivalent functionality available. This type of change doesn't fit the requirements for the other levels of adaptation because the same functionality is not present in the new version of the API.

Level of adaptation	Change type
Cosmetic	Rename function
Composition	Split function
Contract change	Change post condition
Missing	Remove class

Table 4.1.: Examples of change types and their level of adaptation

The complete list of refactorings, non-refactorings and their level of adaptation is available in the appendix in chapter [Change categories](#).

Concrete refactorings or changes may require less adaptation than would be expected from their type. For instance, some specific function may be removed but easily be substituted by another function.

Some classes or functions are mentioned several times. They are changed in multiple ways. This means that the order in which changes will be applied matters. If a function is renamed and changed in another way, it's hard to make sure that all changes are done

when the function is not available anymore with the old name.

The change descriptions are saved in a comma-separated file (CSV). Each line corresponds to one change.

### 4.2.1. Choices

Obsolete functionality is assumed to be unusable, as explained in chapter [Context](#). In Qt 3, obsolescence is indicated by marking functionality as obsolete and not changing its name or location.

C++ function overloading is ignored. A change in one overloaded function is treated as a change in all functions with that name. This choice results in false positives. Source code may seem to be impacted by a certain change when it uses an overloaded function that itself is unchanged. Taking into account overloaded functions would make the analysis more complicated and would likely result in false negatives because of simple typos.

### 4.2.2. Results

The changes derived from the documentation are summarized in tables [4.2](#) and [4.3](#).

For example, the Qt documentation contains the following text. In the survey, *QCursorShape* and *ArrowCursor* in the global namespace are marked as being changed. The change type is *move global definition to namespace*.

Qt 3.x is namespace clean. A few global identifiers that had been left in Qt 2.x have been discarded.

Enumeration `Qt::CursorShape` and its values are now part of the special Qt class defined in `qnamespace.h`. If you get compilation errors about these being missing (unlikely, since most of your code will be in classes that inherit from the Qt namespace class), then apply the following changes:

- `QCursorShape` becomes `Qt::CursorShape`
- `ArrowCursor` becomes `Qt::ArrowCursor`
- ...

It's remarkable that a high percentage (41%) of the API changes is of level *cosmetic*. This suggests that the migration from Qt 2 to 3 may not be very hard. However, the percentage of changes with level *missing* is also high (35%).

Refactorings account for only 43% of API changes. This is much lower than observed in the literature (80%[\[DJ06\]](#)). This may imply that this migration is not typical or that the percentage found in that study is not representative.

Change type	Level of adaptation	Number of occurrences
Encapsulate function	Missing	3
Extract definition	Cosmetic	8
Inline preprocessor directive	Cosmetic	6
Merge overloaded functions	Contract change	1
Move function	Cosmetic	9
Move global definition to namespace	Cosmetic	16
Rename class	Cosmetic	7
Rename function	Cosmetic	41
Rename macro	Cosmetic	9
Replace function with cast	Contract change	1
Replace function with method object	Composition	1

Table 4.2.: Refactorings (Qt 2 to 3)

Change type	Level of adaptation	Number of occurrences
Change invariant	Contract change	1
Change post condition	Contract change	37
Re-implement class	Missing	5
Re-implement function	Missing	2
Remove class	Missing	8
Remove constructor	Missing	5
Remove enum	Missing	1
Remove function	Missing	58
Remove parameter	Contract change	16
Split function	Composition	1

Table 4.3.: Non-refactorings (Qt 2 to 3)

### 4.3. Impact analysis with LICIA

Library Changes Impact Analyzer ([LICIA](#)) uses the change descriptions obtained in the survey, the source code of BIAP and Qt version 2.3.2.

The source code of the program is parsed to extract all references to the API. All those references are checked to see whether they occur in the list of the survey. The old version of the API is required to parse the program (just like it's necessary to compile it).

LICIA uses the Elsa[[WCM10](#)] C++ parser to obtain the information from the source code. The parser outputs two types of information.

First, Elsa outputs a typed [abstract syntax tree \(AST\)](#) which contains all references to the Qt API and where they are made. For example, it shows each time a Qt class is constructed. This is necessary to determine which changed functionality is used and which isn't.

Second, Elsa outputs class inheritance graphs in the DOT language[[Gra10](#)]. This is necessary to determine which classes are changed because they inherit from a changed class.

The process of the impact analysis is summarized in chapter [Research method](#) and its steps are described below in the order of execution.

### 4.3.1. Preprocessing

The Elsa parser expects source files to be preprocessed. Preprocessing transforms a source code file by replacing macros with what they stand for. For instance, an include directive is replaced by the contents of the header file that it points to.

### 4.3.2. Extract references to the API

This step analyzes which functionality of the library is used by the source code.

Elsa analyzes the source code and generates a typed [abstract syntax tree \(AST\)](#). The typed abstract syntax tree contains information about the source code. Each variable and invocation is resolved to a certain type. Each type is resolved to a certain header file and line number where it's declared.

Not all references to header files are relevant. References that don't point to the header file of the API are ignored, because we are only interested in functionality of the API. Those references that are made within the library itself are ignored, because the manufacturer of the library adapts it to a newer version.

References made from code not belonging to the library are assumed to be relevant. Also when they are made in another file than the current source code file. Otherwise impact on header files would not be measured.

Relevant references with type *field* (function invocations), *var* (accessing variables) and *ctorVar* (constructor invocations) are saved. Those type names are used in the abstract syntax tree.

The source code location where a reference occurs is saved and so is the location of the declaration in a header file to which it points. These two locations are mentioned in the abstract syntax tree on subsequent lines.

### 4.3.3. Inheritance paths

We assume that a change in a superclass has an effect on its subclasses because the functionality they inherit is now changed. This means that a change in a superclass of a class used by the program has effect on the program.

This step constructs an inheritance hierarchy of all classes used in a certain source file and computes which of those classes are subclasses of other classes.

Elsa can output inheritance hierarchies as graphs in the DOT language[[Gra10](#)]. Each graph describes the relations of one class with its superclasses.

The inheritance relations occurring in the graphs are stored in a two-dimensional array. The indices represent classes and the value represents whether the second class inherits from the first. For instance,  $inheritancePaths[Animal][Cat] == 1$  indicates that Cat inherits from Animal.

The transitive closure of the inheritance paths is calculated to find all reachable paths. This is done with the Floyd-Warshall algorithm[[Flo62](#)]. The result is that indirect paths

are also stored in the array.

#### 4.3.4. Impact analysis

The next step is to check which of the changes from the survey have an effect on the program. The result is a list of code locations that have impact.

This step uses the change descriptions obtained in the survey of API changes, the inheritance paths obtained in the previous step and the references to the API (declarations) obtained before.

For each change, all references to the API are checked to see whether they refer to it. This is done in two or three steps, depending on the change.

First, a list of candidate classes is constructed. These are the API classes on which the change has an effect. The list includes the class that is mentioned in the change. It also includes all classes that inherit from that class. If a change does not have an effect on one particular class, all API classes used by the program are candidate classes.

Second, for each candidate class, all references to the API are checked to see whether they refer to this class. If yes, we may have found a change that has an effect on the program and a specific location in the program where it has this effect.

A reference refers to this class if it refers to the header file in which this class was defined. We assume that this is true if the reference refers to a header file with the same name as the class. In Qt, this assumption holds in general. For example, *QWidget* is defined in the header file *QWidget.h*. If this assumption would not hold, references should be checked in a different way.

Third, if the class is only changed partly, the reference found in the previous step should refer to the part that is changed. For example, only one function in a class may be changed. If the reference does not refer to that function, it has no impact.

If the class is changed completely or the reference refers to the part that is changed, then the change has impact and is added to a list.

## 4.4. Results

The result of the impact analysis is a spreadsheet with code locations. This looks like table 4.4. The first columns are copied from the survey. The last columns, starting with *source file*, point to specific code locations on which the change has an effect. The last column, *used class*, indicates the class used by the source code that caused the impact analysis to find this location. This can be a different class than the class that was changed, namely a class that inherits from the class that was changed.

A programmer can use the spreadsheet to review the code locations that were found and decide how to do the migration.

The impact analysis results in 519 code locations (change suggestions). 20 API changes of 11 different types have impact on BIAP. See table 4.5 for the number of code locations per change type. 7 individual API changes account for all code changes of level *missing*.

## Chapter 4. Experiment

change id	level of adaptation	change type	class	identifier	source file	line	used class
183	contract change	change post condition	QListBox	insertItem	QBrainAtlas.cpp	603	QListBox
183	contract change	change post condition	QListBox	insertItem	QGetScanDial.cpp	122	QListBox
183	contract change	change post condition	QListBox	insertItem	QMarkers3DDial.cpp	291	QListBox
...	...	...	...	...	...	...	...

Table 4.4.: Raw results of impact analysis (shortened example)

Change type	Level of adaptation	Number of occurrences
Change post condition	Contract change	44
Extract definition	Cosmetic	152
Inline preprocessor directive	Cosmetic	35
Re-implement class	Missing	60
Re-implement function	Missing	35
Remove class	Missing	7
Remove constructor	Missing	4
Remove function	Missing	5
Remove parameter	Contract change	24
Rename class	Cosmetic	144
Rename function	Cosmetic	9

Table 4.5.: Occurrence of change types in impact analysis (Qt 2 to 3)

The distribution of changes over the levels of adaptation is shown in figure 4.1. Changes of level *composition* are absent. Most changes are of level *cosmetic* (340, 66%). This suggests that the migration may be not so hard.

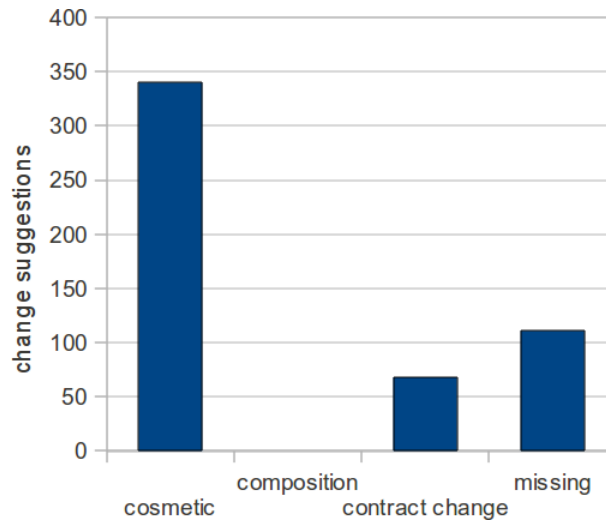


Figure 4.1.: Distribution of change suggestions over levels of adaptation

The changes have an effect on a large number of source files, for each level of adaptation. This means that the migration effort will not be focused on just a handful of files. For example, changes of level *cosmetic* have an effect on 39 source files and changes of level *missing* on 45 source files. There are 15 source files with at least 10 change suggestions.

Changes in the classes *QArray*, *QPainter* and *QPen* account for 64% of the change suggestions. *QArray* is not used directly but is inherited by other classes. In total there are 12 Qt classes that have changes and are used by the program. Some changes are not in classes but in the global namespace.

Refactorings account for 66% of the change suggestions. In the survey, the percentage of refactorings was much lower. Apparently the kind of API changes that have an effect on this program are relatively often refactorings. The two refactorings that occur most frequently (57%) are *extract definition* and *rename class*. The refactoring that occurs most in the survey, *rename function*, occurs only 9 times (2%) in the impact analysis.

## 4.5. Replicating the experiment

The API changes are derived from the API documentation. The documentation should describe which classes, functions and other identifiers are changed and preferably what kind of change it is. If this is impossible, the survey of changes should be done in a different way.

BIAP is written in the C++ programming language. If LICIA is used with source code written in a language that doesn't support inheritance, such as C, the inheritance hierarchy step in the impact analysis can be dropped.

C PreProcessor (CPP) bundled with the GNU Compiler Collection (GCC) version 4.4 is used to preprocess the source code. In the case study, small problems have been found and solved because this preprocessor has a stricter interpretation of the C++ language than the preprocessor normally used with the program (Microsoft Visual C++ 6).

The Elsa parser used in the experiment parses C++ code and outputs an abstract syntax tree with API references and a class hierarchy. It's available on-line with complete source code. Other parsers can generate similar data. For example, ANTLR[Par10] can probably do this for programs written in Java and other languages.

LICIA is implemented as a Perl script. It's available on request at the author.

# Chapter 5.

## Analysis

This chapter describes how the program was migrated to validate the experiment and evaluates the results. The results show whether the techniques are feasible and suggest how they can be improved.

### 5.1. Migrating the program

The complete source code has been migrated. This was done by reviewing all code locations found by LICIA and modifying the source code in those places.

An example of a migration is the change in *QWidget::backgroundColor*. That function is renamed to *QWidget::paletteBackgroundColor*. This change was correctly found in *Q2-ui/QHistFrame.cpp*, *Q2-ui/QFieldView2D.cpp*, *Q2-ui/QLevel.cpp*, *Q2-ui/QColorSelect.cpp* and *Q2-ui/QPlotXY.cpp*. Once in each file. The call to *backgroundColor* on those lines was rewritten to *paletteBackgroundColor*.

All true positives were at most as easy or difficult as suggested by their level of adaptation. For example, there were no cases of *cosmetic* changes that required more than a simple small change.

#### 5.1.1. False negatives

After modifying the source code, it did not yet compile with Qt 3. 17 changes were required to solve the compilation problems. Those 17 changes are false negatives. In most cases the solution was to include a certain header file.

Of those 17 false negatives, 15 were caused by changes not being mentioned in the API documentation. For example, a constructor of class *QFrame* no longer accepts a boolean as its last parameter. This is a change of type *remove parameter* and level *contract change*. The program has been adapted by dropping the last parameter in the constructor invocation.

The other two false negatives are caused by limitations in LICIA.

In the first case, the use of a class wasn't found. The source code contains *QPainter paint(ℰpm);*. This way of referencing *QPainter*, without subsequent use of *paint*, is not mentioned in the typed abstract syntax tree in a way that the script can find. The change that wasn't found is of type *extract definition* and level *cosmetic*.

In the second case, the false negative was caused by the assumption mentioned in section 4.3.4 that a class is declared in a file with the name of the class. *QCursor* was not declared in *qcursor.h* but in *qnamespace.h* in version 2.3.2. In later versions it's in fact declared in *qcursor.h*. The change that wasn't found is of type *extract definition* and level *cosmetic*.

There is insufficient data to conclude that certain types of changes are more likely to cause false negatives than others.

### 5.1.2. False positives

The code was not adapted in all locations that were found by LICIA because in some cases the given change has no effect on the given code location. This means that the code works with the new version of the API without modification. There are 99 such false positives.

53 are caused by one misclassification in the survey of changes. We found the change *re-implement class* for class *QPainter*. This is a human error that may be caused by mistaking *QPrinter* for *QPainter* because of the similarity in name.

28 are caused by the choice mentioned in section 4.2.1 of not distinguishing overloaded functions. LICIA finds uses of the function *QWidget::setFont* and the constructor *QToolBar*. BIAP uses a function and a constructor with those names, but not the overloaded variant that has been changed.

18 are caused by the assumption mentioned in section 4.3.3 that changes in a superclass have an effect on its subclasses. In these 18 cases, the class itself wasn't changed although a superclass was renamed or removed. For example, uses of *QString* have been found and marked as being impacted by the *rename class* change in *QArray*. *QString* inherits from *QArray* via *QByteArray* which explains why LICIA thinks that the change has an effect on *QString*. However, the change has no effect because *QByteArray* now inherits from *QMemArray* which is the new name of *QArray*.

### 5.1.3. Superfluous results

An unanticipated result is that some results are superfluous. They are not false positives, but they don't result in action either. For example, if a certain class is defined in a separate header file in the new version, all uses of that class will generate a change suggestion to add the header inclusion. That action only has to be done once for each source code file. This is the distinction made in section 1.2 of chapter [Introduction](#).

The existence of superfluous change suggestions means that the real number of code adaptations is lower than the number of change suggestions.

In the experiment, changes of type *extract definition* cause superfluous results. The 152 change suggestions result in 55 code adaptations. This means that 97 change suggestions (19%) are superfluous.

### 5.1.4. Success of migration

The program was compiled with Qt version 3.3.7 using the Microsoft Visual C++ 6.0 compiler on Windows XP. It was also compiled with Qt 3.3.8 using GCC 4.4.3 on Ubuntu 10.04.

One problem was found on both systems. The alignment of QSpinBoxes next to QListBoxes is different from the alignment in Qt 2. This change is not mentioned in the documentation.

The other problems only occur on Windows. They can not be traced to differences mentioned in the documentation. This means that the survey and the impact analysis could not have found them. These problems were found on Windows:

- A custom drawing cursor should be displayed in some contexts. It sometimes displays the normal cursor instead of the drawing cursor. The reason is not clear.
- Several widgets, including QListBox, don't correctly repaint. This means that the contents change but the display isn't updated. The reason is not clear.

Apart from this, the program works as expected. This means that the migration is successful. There is no test suite available so the program was tested by performing operations and evaluating the result.

### 5.1.5. Precision and recall

The rate of false positives (type I errors) and false negatives (type II errors) are shown in table 5.1. This means that precision is 81% and recall is 96% for this experiment. The number of true negatives is high and hard to determine because in theory each code location can have impact from multiple changes.

	Change has impact	Change has no impact
LICIA finds impact	True Positive: 420	<i>False Positive: 99</i>
LICIA finds no impact	<i>False Negative: 17</i>	True Negative: Unknown

Table 5.1.: Type I and II errors

### 5.1.6. Conclusion

The migration was easier than suggested by LICIA because not all code locations that were found had to be adapted and the changes that were not found were not hard. Most changes were of level *cosmetic*, as suggested by LICIA.

The number of changes of level *missing* was lower than suggested by LICIA because most false positives are of that level and there are no changes of that level that were not found.

## 5.2. Threats to validity

Deriving the API changes from the Qt documentation and writing them down in a list is a manual process. This introduces the risk of human errors. This risk is limited by verifying a random set of 50 changes. Verification is done by looking up the change in the API documentation and determining whether the functionality is changed in the way that is stated in the survey.

Using the Qt documentation to make the survey of changes introduces the risk that the survey is incomplete if the documentation is incomplete. The completeness is validated by adapting the program for those changes that are found and determining whether this results in a successful migration.

The levels of adaptation that are proposed and used in this study are based on how hard it's expected to be to adapt the program to a change of a certain type. This expectation is based on programmer's intuition and not validated by for instance measuring the amount of time spent on adapting the program. In the experiment, true positives were at most as easy or difficult as suggested by their level of adaptation. For example, there were no cases of *cosmetic* changes that required more than a simple small change.

The feasibility of the techniques is tested with only one experiment. This means that the results may not apply generally for all programs that are migrated to a newer version of a library. Section 6.1 addresses this issue in detail.

Changes may not have been found if the old functionality is marked obsolete in the new version. This means that there may be more false negatives than found during the experiment. Section 4.2.1 describes that obsolete functionality is considered to be unusable. This means that changes in which old functionality is available as obsolete functionality in the new version should be found by LICIA. However, if the program uses obsolete functionality it's likely that it works normally and that the migration appears to be successful. The reason is that use of obsolete functionality may not generate warnings by the compiler or at run-time. Consequently, certain changes may not have been found by LICIA that should have been found and this problem may not have been found either.

## 5.3. Implications for the techniques

LICIA produces an overview of code locations on which API changes have an effect with a precision of 81% and a recall of 96% in the experiment.

The quality of the survey depends very much on the source that is used to find API changes. In the experiment, 15 of the 17 false negatives are caused by API changes not mentioned in the documentation of Qt.

The quality of the impact analysis depends very much on the quality of the survey of API changes. In the experiment, one human mistake causes 54% of the false positives.

## 5.4. Improvements

The limitations and assumptions in the techniques account for 46 false positives (46%) in the experiment.

Section 4.2.1 describes the choice of not distinguishing overloaded functions. 28 false positives can be avoided by using the full type information of functions and constructors and using it to treat overloaded functions as different functions. For example, if a constructor `MyClass::MyClass(int value)` is changed, the survey should mention not just that a constructor of class `MyClass` is changed but that the constructor that takes one `int` variable is changed. The impact analysis should use that information to check whether exactly that constructor is used instead of another constructor of the same class.

Section 4.3.3 describes the assumption that changes in a superclass have an effect on its subclasses. 18 false positives can be avoided by taking into account which changes have an effect on subclasses and which don't. Apparently the assumption does not hold generally. For example, if class A has a change of type `rename class`, class B inherits from class A and a program uses class B, this change has no effect. Class B itself should be adapted to the change in class A, but the manufacturer of the library will do that work. The public interface of class B doesn't change.

Only one false negative could have been avoided in the experiment by improving LICIA. It could have been avoided if the impact analysis would parse all header files instead of assuming that a class is defined in a header file with the same name. This assumption is mentioned in section 4.3.4.

The number of superfluous results could be reduced by taking into account the change type in the impact analysis. In the experiment, changes of type `extract definition` cause superfluous results. Changes of that type have an effect on a source file at most once, because adding a header file inclusion directive to a source file only has to be done one time. Currently, the analysis doesn't take this into account and reports each use of functionality from that header file as having impact. A drawback of making the impact analysis aware of the properties of change types is that this makes the impact analysis more complicated.

# Chapter 6.

## Conclusion

The problem studied in this report is that programmers don't know how complicated an API migration of a program is before it's finished. This means that they cannot choose an appropriate migration strategy.

The research question is "How can the impact of API evolution on a program be measured automatically?". The hypothesis is that LICIA produces an overview of all code locations on which API changes have an effect.

The number of code locations in the program on which API evolution has impact is a measure for the impact of API evolution on a program. LICIA finds those code locations in two steps. First, the API changes are manually looked up and written down in a certain format. Second, the program is automatically analyzed to find references to parts of the API that are changed. The result is a spreadsheet that shows which source code lines in the program should be adapted and which changes have an effect on them.

The experiment done to assess the feasibility of the solution is the migration of a brain image analysis program written in C++ to a newer version of the Qt Framework. This experiment has been chosen because the program and the framework have a size and complexity that is comparable to those found in real migrations and because enough information is available to conduct the experiment.

The migration is successful. Precision is 81% and recall is 96%. 19% of the results is superfluous. The false negatives are mostly caused by API changes that are not mentioned in the documentation. The false positives are mostly caused by one mistake in the survey and two assumptions in the impact analysis. The superfluous results are caused by not taking into account the specific change type in the impact analysis.

LICIA doesn't produce an overview of all code locations on which API changes have an effect, as stated in the hypothesis, because recall is not 100%.

### 6.1. Implications for other migrations

There are several reasons why LICIA may not work well for other migrations.

First, the quality of the API documentation may be lower. In this experiment, API changes are derived from the documentation provided by the manufacturer of the API. That documentation gives the impression of being of high quality. However, most false negatives are caused by API changes not being mentioned in the documentation. That

means that the results depend very much on the quality of the documentation. If the API changes are not well documented, the quality of the impact analysis will be low.

Second, there may be certain types of changes that LICIA cannot detect. In the experiment, 11 change types have an effect on the program. In other migrations, other change types may have an effect.

Third, the changes that break the assumptions in LICIA or that cause superfluous results may occur more frequently in other migrations. That means that the quality of the result will be lower in those migrations. For example, if there are changes that subclasses don't inherit from superclasses or if there are changes such as *extract definition* that require at most one adaptation per source file, LICIA will produce false positives or superfluous results.

On the other hand, if the experiment is representative for another migration LICIA may work very well. There are no known limitations in LICIA that restrict its use to this specific experiment.

No other studies are known that assess the impact of API evolution on a program. That means that the results cannot currently be compared with other studies.

## 6.2. Future work

The limiting factor on the quality of the impact analysis is the quality of the API documentation used in the survey. Manually deriving API changes from the documentation is also the step that costs most of the time. In future work, an automatic process can be developed to deduce the API changes from the two versions of the API. It can be based on techniques described in chapter [Context](#). Such a technique can improve the quality of the impact analysis and make it a fully automatic process.

The number of false positives can be reduced by reviewing the assumptions and handling the cases in which they don't hold. For example, the assumption of section [4.3.3](#) that subclasses inherit changes from their superclasses causes false positives for some change types such as *rename class*. The solution can be to add a list of change types that don't have an effect on subclasses to LICIA and take this into account when constructing the list of candidate classes during the impact analysis step.

Many changes can be seen as refactorings but only a small number of them is well described in the literature. This study proposes names for refactorings that have no commonly used names. Those refactorings can be described more formally in future work. Having a complete list of refactorings is useful to make communication among computer scientists easier.

The levels of adaptation that are proposed and used in this study merit further research. It would be especially interesting to validate that changes with a higher level of adaptation require more effort than changes with a lower level of adaptation. Required effort can be measured in terms of time spent or number of actions performed. The levels of adaptation are useful to indicate in a concise way how many changes there are and how hard they are.

The impact analysis tells exactly which source code line should be adapted to which

change. It would be interesting to take the next step and make some of those changes automatically. This is probably only possible for *cosmetic* refactorings. Whether it helps much depends on the specific migration; in this case most changes could have been made automatically.

We predict that LICIA or a technique based on LICIA will be integrated in Integrated Development Environments (IDE). Programmers will be able to use their IDE to advise them about API migrations. When they want to migrate a program to a newer version of a library, the IDE shows exactly which source code lines should be adapted and why. The IDE will propose to do most of the changes automatically. If the migration appears to be complicated, programmers can make a plan based on the amount and type of changes that have an effect on the program they work on. The IDE will show them this information graphically and in tables. API migration can be a much more pleasant and less error-prone process than it currently is.

# Glossary

## A

**abstract syntax tree (AST)** An abstract syntax tree is a representation of the syntactic structure of the source code of a program. It can be used to derive information from source code. For example information about the type of a certain variable. Abstract syntax trees are not used to compile a program and they may not contain all the information present in the source code. In the context of the Elsa parser, an AST does not contain all type information while a typed AST does. (See page 9)

**API migration** The process of modifying a program such that it doesn't use a particular library or framework anymore, but uses a different one instead. This replacement library or framework exposes its functionality with a different API. The replacement library or framework can be unrelated to the previous one (developed by a different company, using different semantics). Usually it is a newer version of the library or framework it replaces; the APIs are essentially the same. (See page 4)

## B

**BIAP** Brain Image Analysis Package (BIAP)[dM10] is the program used in the experiment. It can be used to analyze, segment and visualize images from CT, MRI, PET, MEG and other measurements. BIAP is developed at the department Physics and Medical Technology at the VUmc university hospital. (See page 2)

## D

**deprecate-replace-remove cycle** Cycle related to software evolution. See also [obsolete functionality](#). Old functionality is deprecated (marked obsolete), replaced by new functionality and one or more versions later removed. (See page 5)

## L

**level of adaptation** In this study, the level of adaptation is a measure for how hard it is to adapt a program to a certain change in the framework/library it uses. A change with a higher level of adaptation is expected to require more

## GLOSSARY

complicated adaptations and/or adaptations in more locations in the source code of the program. (See page 8)

**LICIA** LLibrary Changes Impact Analyzer (LICIA) is the technique studied in this report. It is a technique for finding how changes in an API have an effect on a (big) program that uses it. It produces a list of code locations that should be adapted in a program, together with information about what changes have an effect on those locations. These results provide information about which source files are most affected by changes and which API classes cause most of the changes. This allows a programmer to know the difficulties in a migration in advance and adapt the migration strategy to the data. (See page 2)

## M

**major version** If a version is numbered 2.3, the major version is 2 and the minor version is 3. Major versions are usually not backwards compatible. (See page 10)

**minor version** See also [major version](#). In the context of Qt, minor versions with the same major version are backwards compatible. For instance, a program made for Qt 2.0 will work with Qt 2.3. (See page 10)

## O

**obsolete functionality** Obsolete functionality is functionality that has already been replaced by new functionality but is temporarily kept in code. This makes the change process more graceful for programmers. Migration can be done in steps. Code that uses the changed functionality does not break but programmers are strongly suggested to update. This process is called the “deprecate-replace-remove” cycle in the literature. (See page 5)

## R

**refactoring** Refactorings are “program transformations that change the structure of a program but not its behavior” [DJ06]. (See page 5)

## V

**virtual function** C++ supports dynamic binding of functions with the keyword *virtual*. If this keyword is used, it is not possible to statically determine which function will be called at run-time because this decision is postponed. Which function is chosen depends on the dynamic type of the object on which it is called. (See page 10)

# Bibliography

- [BCLvdS10] T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API migration for two XML APIs. *Software Language Engineering*, pages 42–61, 2010.
- [CN96] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 International Conference on Software Maintenance*, page 359. IEEE Computer Society Washington, DC, USA, 1996.
- [DJ06] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.
- [dM10] J. C. de Munck. Brain image analysis package, 2010. <http://demunck.info/software>.
- [DR08] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th international conference on Software engineering*, pages 481–490. ACM, 2008.
- [FBB<sup>+</sup>99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.
- [Flo62] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [Fow10] Martin Fowler. Alpha list of refactorings, July 2010. <http://www.refactoring.com/catalog/index.html>.
- [Gra10] Graphviz. DOT language, 2010. <http://www.graphviz.org/doc/info/lang.html>.
- [LR01] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.
- [Par10] Terence Parr. ANTLR, July 2010. <http://www.antlr.org>.
- [TDX07] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 377–380. ACM, 2007.
- [Tro02] Trolltech. Porting to Qt 3.x, 2002. <http://doc.qt.nokia.com/3.0/porting.html>.
- [Tro10] Trolltech. The Qt 3 to 4 Porting Tool, 2010. <http://doc.qt.nokia.com>.

## Bibliography

[com/4.6/qt3to4.html](http://danielwilkerson.com/4.6/qt3to4.html).

- [WCM10] Daniel S. Wilkerson, Karl Chen, and Scott McPeak. Oink stack, 2010. <http://danielwilkerson.com/oink>.
- [WGAK10] W. Wu, Y. G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A Hybrid Approach to Identify Framework Evolution. In *ICSE '10*, May 2010.
- [XS07] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.
- [ZTX<sup>+</sup>10] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API Mapping for Language Migration. 2010.

# Appendix A.

## Migration advice

The migration from Qt 2.3.2 to Qt 3 was done during the experiment. The ultimate goal is to migrate the program from Qt 2.3.2 to Qt 4, the most recent version. This chapter provides advice for the last step, from Qt 3 to Qt 4.

The advice in this chapter is based on the results of the study. Section 6.1 describes what the results of the study imply for other migrations such as the migration from Qt 3 to 4. Section [Analysis](#) addresses the implications for the advice.

### A.1. Strategy

There are three migration strategies.

The first is to completely migrate the program in one go, using the impact analysis to know what should be adapted. This strategy is the least work but it's also the hardest because the program will only compile when the migration is finished.

The second strategy is to migrate to the Qt 3 compatibility layer of Qt 4 first and then on to pure Qt 4. Migrating to the compatibility layer is much less work than completely migrating. Migrating from the compatibility layer to pure Qt 4 is easy because it can be done class by class. Halfway during the migration the program will work. When everything is migrated to pure Qt 4, the compatibility layer can be turned off.

The third strategy is to adapt the program while staying with Qt 3<sup>1</sup> and then start the actual migration. The same functionality can be implemented in Qt 3 in several ways. Some of those ways are still supported in Qt 4 while others are not. Using the way that does not change makes the migration to Qt 4 easier. This strategy is similar to the compatibility layer strategy but has the benefit of having a compilable program after each small step.

The above strategies can be combined. The easiest migration would be to use all of the strategies together. First, adapting the program while staying with Qt 3, then migrating to the compatibility layer and then using the impact analysis to do the last step.

---

<sup>1</sup>First adapting the program while staying with Qt 3 in order to make the migration to Qt 4 easier is described in <http://doc.qt.nokia.com/4.6/porting4-overview.html>

## A.2. Survey of API changes

The refactorings and other changes occurring from Qt version 3 to 4 are summarized in tables [A.1](#) and [A.2](#). See chapter [Change categories](#) in the appendix for definitions of the names.

It's remarkable that there are much more changes than from Qt 2 to 3, 1238 versus 202. It's also remarkable that 57% is of the highest level *missing* and only 30% of the lowest level *cosmetic*. This suggests that this migration is much harder than the migration from Qt 2 to 3.

Change type	Level of adaptation	Number of occurrences
Change default value of property	Contract change	1
Encapsulate function	Missing	292
Encapsulate property	Missing	6
Move enum value	Cosmetic	64
Move function	Cosmetic	1
Move property	Cosmetic	1
Move type	Cosmetic	12
Rename class	Cosmetic	5
Rename enum value	Cosmetic	160
Rename function	Cosmetic	25
Rename macro	Cosmetic	2
Rename property	Cosmetic	31
Rename type	Cosmetic	14
Replace class with function calls	Composition	1
Replace constant with function call	Contract change	3
Replace default behavior with property	Composition	1
Replace explicit with implicit data sharing	Contract change	3
Replace flag with attribute	Composition	5
Replace function with constructor parameter	Composition	1
Replace function with signal	Composition	6
Replace functions with single event handler	Composition	6
Replace parameter with explicit functions	Composition	1
Replace property with different property	Composition	8
Replace property with multiple function calls	Composition	3
Rewrite enum value	Cosmetic	1
Unmark function as slot	Missing	12

Table A.1.: Refactorings from Qt 3 to 4

## A.3. Impact analysis

The impact analysis is done both for the migration to the Qt 3 compatibility layer and for the migration to Qt 4 directly. Without the compatibility layer there are 9255 change suggestions. With it there are 8407 change suggestions. Especially the relative number of level *missing* changes is significantly lower with the compatibility layer (645 less). See figure [A.1](#).

The impact analysis generates false positives. Changes may not have an effect at all because a function or class may be used in a way that is not changed. For example, the

Change type	Level of adaptation	Number of occurrences
Add parameter to constructor	Contract change	1
Add parameter to function	Contract change	4
Change post condition	Contract change	37
Change pre condition	Contract change	30
Change type	Contract change	3
Re-implement class	Missing	63
Remove class	Missing	88
Remove constant	Missing	15
Remove constructor	Missing	2
Remove enum value	Missing	11
Remove function	Missing	128
Remove functionality from function	Missing	2
Remove parameter	Contract change	11
Remove property	Missing	26
Remove type	Missing	2
Restrict use of class	Missing	1
Unclassified	Missing	7
Unclear changes in functionality	Missing	3 classes, 2 functions

Table A.2.: Non-refactorings from Qt 3 to 4

change *encapsulate function* of level *missing* is found 563 times while the change only has an effect if the function is (re)defined in an inherited class.

The distribution of change types of level *missing* can be seen in figure A.2. Those changes have an effect on a large number of source files. There are 12 source files in which at least 100 change suggestions are given. Those source files are all in the *Q2-ui* directory.

82% of the change suggestions are related to changes in only 6 Qt classes, if the Qt 3 compatibility layer is used. The classes are *QPushButton*, *QFrame*, *QLineEdit*, *QRangeControl* (inherited by *QSpinBox*), *QString* and *QWidget*. Knowing how these classes are changed and how they should be used now makes it possible to do most of the migration. This means that the migration may be easier than it seems.

If the Qt 3 compatibility layer is used, most *cosmetic* changes can be done automatically. A script is provided with Qt 4 that adapts source code to renamings[[Tro10](#)].

## A.4. User Interface files

The impact analysis only processes C++ source code files. Another type of file used in Qt is the User Interface (UI) file.

UI files are used to design the graphical user interface. They are created and edited with the Qt Designer tool. They are used to generate C++ ‘Base’ classes. Those classes are subclassed to specify the behavior and link them to the rest of the program.

Qt Designer is still used in Qt 4 but has less functionality. The scope of UI files is also smaller. The Qt documentation is not specific about what changes exactly.

Converting UI files from Qt 3 to Qt 4 is possible. A tool is provided with Qt 4 to do

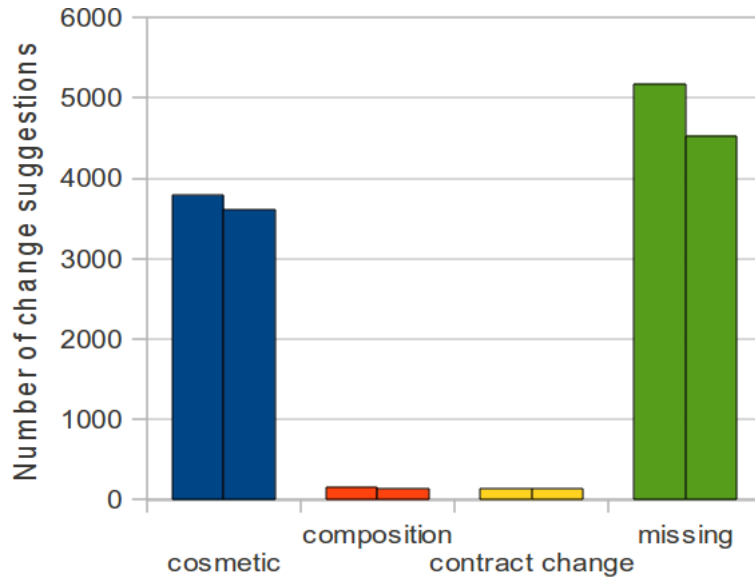


Figure A.1.: Change suggestions without Qt 3 compatibility (left) and with (right)

this. Custom icons and signals and slots<sup>2</sup> disappear because they are not supported in Qt 4 anymore. This is not a problem for this program because it does not use that functionality.

The source code of the program should interact with generated ‘Base’ classes in a different way. First, the class name has changed and is now prefixed with the *Ui* namespace. For instance, a class name may be changed from *OpenFileBase* to *Ui:OpenFileBase* or *Ui:OpenFile*. Second, in addition to the ‘Base’ class, the program should now also inherit from *QWidget* (or one of its subclasses if applicable). Third, the constructor should now also call *setupUi()* to initialize the UI it inherits.

## A.5. Analysis

The advice in this chapter is based on the results of the study. Section 6.1 describes what the results of the study imply for other migrations such as the migration from Qt 3 to 4.

Three reasons are mentioned why LICIA may not work for other migrations. These reasons are analyzed for this migration.

First, the documentation for the migration from Qt 3 to 4 may have a lower quality than the documentation for the migration from Qt 2 to 3. If the documentation is of lower quality, the result of the impact analysis will also be of lower quality.

Second, there are many changes that occur from Qt 3 to 4 that did not occur in the experiment. There may be changes that LICIA cannot correctly identify. For example, *encapsulate function* is a change that is found frequently in this migration (563 times)

<sup>2</sup>Signals and slots is a feature of Qt used for sending information through a program. Signals are the information being sent. Slots are the receiving functions of that information.

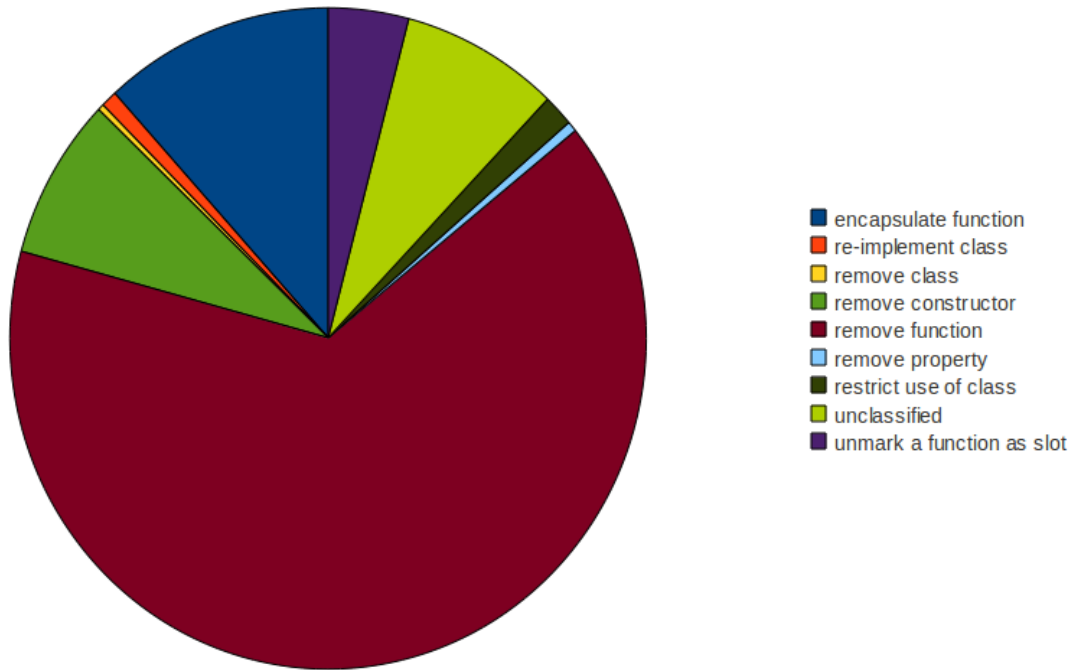


Figure A.2.: Suggestions per change type of level *missing*, with Qt 3 compatibility layer

and that also has a high level of adaptation (*missing*). This change means that a certain function can no longer be redefined in subclasses. This does not break the assumption of section 4.3.3 that changes in a superclass have an effect on its subclasses, but it reveals that some changes have an effect on subclasses *without* having an effect on uses of the class itself. This is likely to cause false positives.

Third, some changes cause false positives or superfluous results in the experiment. Such changes may occur more frequently in this migration. Changes of type *remove function* occur frequently (128 times) and can cause false positives if not all functions with the same name are removed because of the choice described in section 4.2.1 not to distinguish overloaded functions. However, the change type that caused all superfluous results in the experiment, *extract definition*, does not occur in this migration.

## A.6. Advice

We recommend to do the following steps to migrate the program.

First, make the program more robust by using a stricter compiler (such as GCC) and reviewing the warnings it emits. Many warnings point to weak code. Such weaknesses may currently not cause problems but they can result in strange behavior if changes are made to the program.

Second, use a version management system such as Subversion to manage the source code. The migration will be complex and it will be important to know exactly which code was changed when. Errors will be made and a version management system makes it much easier to backtrack them.

## Appendix A. Migration advice

Third, adapt the program while staying with Qt 3. This specifically applies to the *QMenuBar* and *QMenuData* classes which can be rewritten to use *QAction*.

Fourth, migrate to the Qt 3 compatibility layer of Qt 4. Start by running the Qt 3 to 4 porting tool. It may be best to choose a part of the program that can be build independently and migrate only that part first. *QView2D* would be suitable for this. The smaller the amount of code to migrate, the more the migration can be kept under control.

Fifth, migrate the parts that use the Qt 3 compatibility layer one at a time and check the result after each step. The compatibility layer can be disabled when it's not used anymore. At that moment the migration is complete.

# Appendix B.

## Change categories

This chapter lists all change types found during the experiment and during the survey of chapter [Migration advice](#). See chapter [Experiment](#) for further information.

Refactorings that are described in the literature contain a reference to that literature. Methods and functions are called functions in this study.

### Level 0: Compatible change

Backwards-compatible change. These are not mentioned.

### Level 1: Cosmetic

The same functionality is present and only a simple small change is required to use it. For example: name change, including an additional header file, moving an enum to a different class.

### Refactorings

- Extract definition: Move the definitions of a class to its own header file from a big header file that contains more definitions. The new header file corresponds with the name of the class being defined.
- Inline preprocessor directive: Remove a preprocessor directive and replace its usages with what the compiler would have substituted.
- Move enum value[Fow10]: Move a certain enum value to a different enum. (In the literature the more general name *Move field* is used.)
- Move function[Fow10]: Move a function from one class to another. Moving within a class is considered a renaming. Special case: Mark function as static; i.e. move the function from an instance to the class.
- Move global definition to namespace: Restrict the scope of a definition by moving it from global scope to a namespace.

- Move type[Fow10]: Move an enum or typedef to a different class. (In the literature the more general name *Move field* is used.)
- Rename class: Rename a class name and its associated header file name. Note that C++ doesn't enforce updating the header file name when the class name changes.
- Rename enum value: Give an enum value a different name but do not change the name of the enum itself.
- Rename function[Fow10]: Rename a function while keeping it in the same class. A special case of renaming a function may be renaming a global function because it may be harder to determine what to change.
- Rename macro: Rename a macro variable or function. In this case, the macros are only used for debugging output to the console.
- Rename property: Rename an attribute of a class and its associated getter and setter functions.
- Rename type: Give an enum or typedef a different name but keep it in the same place.
- Rewrite enum value: Represent an enum value in a different way, such as the result of a function call.

## Level 2: Composition

The same functionality is present but some extra calls are required to use it. The contract did not change so no pre or post processing is needed.

### Refactorings

- Move property: Move an attribute of a class and its associated getter and setter functions to another class. (This is almost the same as *Move field*[Fow10]. They don't use getter and setter functions.)
- Replace class with function calls: Certain functionality that used to be performed with a class. The same functionality can now be obtained by calling one or more functions.
- Replace default behavior with property: Change the default behavior and introduce a property to get the same behavior back.
- Replace flag with attribute: Replace an enum value used as flag passed to an object with an attribute of that object.
- Replace function with constructor parameter: Instead of allowing a property to be set with a function, let it be passed to the constructor.
- Replace function with method object[Fow10]: Move functionality from a function to a separate object.
- Replace function with signal: Instead of (callback) functions use signals. This is

related to Qt signals and slots.

- Replace functions with single event handler: Replace several (virtual) functions that are called when an event occurs with a single event handler function.
- Replace parameter with explicit functions[Fow10]: Functionality used to be performed when passing a parameter to a function or constructor. Move that functionality to a separate function.
- Replace property with different property: Implement the functionality of a property in a different property (of the same or another class). Old property and associated getter and setter functions are removed.
- Replace property with multiple function calls: Instead of performing an operation automatically if the property is set, the operation should be called explicitly.

### Non-refactorings

- Split function: Split the functionality of a function into multiple functions.

## Level 3: Contract change

The same functionality is present but some extra code may be necessary to use it. In addition to level 2, the extra code may do pre or post processing. For example: parameter added to function, different return type.

### Refactorings

- Merge overloaded functions: Use a variable argument list to define one function that performs the same operation as two old functions. The order of the argument list may change.
- Replace constant with function call: Instead of providing a certain value as a constant, provide it as the result of a function call.
- Replace function with cast: If a cast can perform the same operation as a function, the function is considered to be superfluous and can be removed.
- Replace explicit with implicit data sharing: No longer use shared data by default, automatically creating a separate object when it's changed.
- Change default value of property: Change the default value of a property to a different value. In order to keep the same behavior, the desired value should now be set.

### Non-refactorings

- Add parameter to constructor[Fow10]: Require an additional parameter in a constructor. (In the literature called *Add parameter* and only used for functions.)

- Add parameter to function[Fow10]: Add a parameter to the parameter list of a function. (In the literature called *Add parameter*.)
- Change invariant: Affect the environment of a code unit in a different way. For instance, a member function may alter instance variables that it didn't alter before and this change may result in a different behavior of the system.
- Change post condition: Change the return type, its valid range of values or other guarantees about its value.
- Change pre condition: Change the type or valid range of parameters of a function.
- Change type: Change the type of a typedef.
- Remove parameter[Fow10]: Remove a parameter from the parameter list of a function. This is similar to removing an overloaded function. (In [Fow10] this is considered a refactoring. Here it's not because the removed parameter may imply that certain functionality is no longer present.)

## Level 4: Missing

The same functionality is not present and cannot be achieved by composing functions (like level 4 in [BCLvdS10]). A significant amount of new code may be needed to reproduce the same behavior with the new library or it may be impossible to do so.

In a special case, functionality may have been removed from the interface because it's now performed automatically by the library. The behavior may be slightly different.

## Refactorings

- Encapsulate function: Mark a function as non-virtual so children cannot redefine it and its functionality is encapsulated in the parent.
- Encapsulate property[Fow10]: Make a property no longer directly accessible without using the getter and setter functions. Keep or introduce getter and setter functions. (The literature uses the more general name *Encapsulate field*.)
- Unmark function as slot: No longer mark a function as being a slot. This is related to Qt signals and slots.

## Non-refactorings

- Re-implement class: Rewrite a class such that the differences are too big to see individually and code using the class should also be changed drastically.
- Re-implement function: Rewrite a function such that the differences are too big to see individually and code using the function should also be changed drastically. The new function has the same name.
- Remove class: Completely remove a class and don't replace its functionality. A special case is when the class would normally only be used internally in the library.

## *Appendix B. Change categories*

- Remove constant: No longer provide a certain constant in a namespace or class.
- Remove constructor: Remove a constructor from a class. By definition another constructor remains available.
- Remove enum: Remove an enum and don't replace it with something else.
- Remove enum value: No longer provide a certain value in an enum.
- Remove function: Completely remove a function and don't replace it with an alternative. In some cases the functionality is now performed automatically in the library (encapsulated).
- Remove functionality from function: No longer support certain parameter values, or move the functionality to another function or class.
- Remove property: Completely remove a property and its associated getter and setter functions.
- Remove type: Remove a certain enum or typedef.
- Restrict use of class: Restrict in which parts of the program a class may be used.
- Unclear changes in functionality: A certain class or function is changed but the documentation is not clear about what changed exactly.
- Unclassified: A certain class or function is changed and the documentation describes how. However, the change cannot be described in a general way in several words.