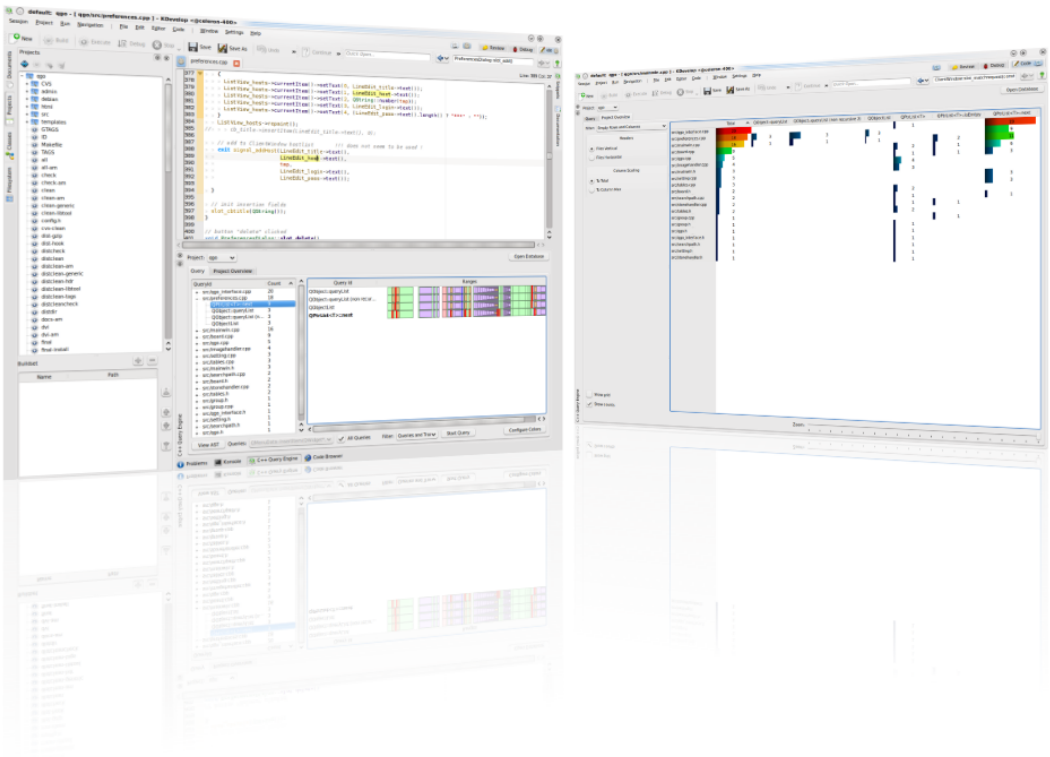
A VISUAL TOOL-BASED APPROACH TO PORTING C++ CODE.

BERTJAN BROEKSEMA



Master thesis

Rijksuniversiteit Groningen
Faculty of mathematics and natural sciences
Department Computing Science

Supervisor:
Prof. dr. A. C. Telea
KDAB Supervisors:
T. Adam
V. Krause

Bertjan Broeksema: *A Visual Tool-Based Approach to porting C++ Code.*, Master thesis, © June 2010

ABSTRACT

The sheer number of changes needed to port a code base when one of more of its dependencies need to be replaced by either a new version or another framework make it viable to develop tools that perform all or most of the required changes automatically in a reliable way. The purpose of this thesis was to research ways to support developers in the process of automated refactoring of large code bases.

To this extent the type of refactoring activities involved was studied. Furthermore, a semi-automated code refactoring system based on queries and rules was developed. Transformations work directly on the source by means of insert and replace actions but are still correct due to the semantic understanding of the code by the framework.

Additionally, new visualization techniques are proposed to support the process of iterative refactoring. The results of these are implemented in a tool for the C/C++ language, developed as an extension of the KDevelop Integrated Development Environment (IDE). Finally, the effectiveness and usefulness of the tool was demonstrated on a large industrial code base and concrete refactoring operations involved in the process of porting C++ code.

The tool is able to deal with large and complex code bases. It was tested with a subset of the queries and transformations needed for a Qt3 to Qt4 port on kdelibs 3.5 which contains about 750K lines of code. The project overview visualization gives a clear overview of the results for the queries on the project and can easily show results for up to hundreds of files. The file impact visualization gives a space efficient overview of the structure of a source file and at the same time it gives clear visual hints of potential conflicting changes. Finally, transformations were specified for about fifteen of the more complex changes in a Qt port.

The approach taken for transformations worked particularly well when changes are localized. For more structural changes to the code transformations on the Abstract Syntax Tree (AST) and AST pretty printing would be required. The project overview visualization is really helpful in estimating porting effort. Finally, the file impact visualization extended the infrastructure by enabling code complexity analysis in addition to the main task of automated refactoring.

ACKNOWLEDGMENTS

First I'd like to thank prof. dr. Alex Telea who supervised the work done for this thesis. His expertise in the field of C++ fact extraction as well as his support during the writing phase made the project a very interesting and pleasant ride.

I also want to thank KDAB for offering the opportunity to work on this highly interesting topic in a very nice environment. Special thanks go to Till Adam and Volker Krause who supervised my work and who were nice enough to review several versions of this thesis.

Next I want to thank the KDevelop developers for the development of a feature rich and extensible IDE. In particular I'd like to thank Milian Wolff, who picked up my initial implementation and performed numerous stability enhancements.

Finally, I'd like to thank my beloved wife Agnes who was a great support during the whole project, even though she was in the process of writing her own thesis. Your support was invaluable.

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

AOP Aspect Oriented Programming

API Application Programming Interface

AST Abstract Syntax Tree

ASG Annotated Syntax Graph

CAN C++ Analyzer

CANPP C++ Analyzer Preprocessor

DUChain Definition Use Chain

GUI Graphical User Interface

GLR Generalized Left-to-right Rightmost

IDE Integrated Development Environment

KDE SC KDE Software Compilation

MFC Microsoft Foundation Classes

moc meta object compiler

ODR One Definition Rule

Qid Qualified Identifier

SDF Syntaxt Definition Formalism

1 INTRODUCTION

In the typical life cycle of industrial software projects, software maintenance takes up to eighty percent of the overall project costs. Forty percent of these costs are spent on program understanding. Therefore it is desirable to have tools that make the maintenance process more efficient and effective at several levels Lanza et al. [26], Diehl [22].

An important aspect of software maintenance is keeping the code base up to date with respect to a changing environment. Changes in the Application Programming Interface (API) of the projects dependencies can have a large impact on the code base. Porting a code base to a new version of one or more of the dependencies might, depending on the project size, become a huge effort. The actual activity of porting looks related to refactoring. In his book on refactoring, Martin Fowler describes it as the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing new bugs Fowler [25].

For porting on the other hand, the short term result is often a code base that needs a lot of manual work. Another difference is that when porting a code base, many of the needed changes to the code base can be formalized in such a way that makes them suitable for automated tools. In addition to that, the vast amount of changes needed on industry sized code bases makes it economically interesting to develop tools that help to automate the porting Akers et al. [16].

This is a well known fact at KDAB [6] where the work for this thesis was performed. KDAB is a middle size software company that is specialized in Qt [11] development and consulting. Qt is a cross platform application and user interface framework for the C++ programming language. A major part of the engineering work at KDAB consists of porting C++ code bases. Several kinds of ports are done, the most common ones being:

- Microsoft Foundation Classes (MFC) to Qt
- Motif to Qt
- QtX to QtY (with $X < Y$).

Toolkits like MFC [8] and Motif [9] are similar to Qt in that they are application development frameworks providing tools such as generic containers, Graphical User Interface (GUI) widgets, XML

handling, etc. The way that these concepts are expressed in actual API differs between the various toolkits. In contrast, two major versions of the same framework are in many cases very similar. This results in the fact that many of the changes needed in a Qt3 to Qt4 port are local to one method. MFC ports however, also require a lot of changes to the logic of the code. For Motif ports it is even more complex due to some missing concepts in the Motif API which are required in Qt based code. In practice, KDAB discovered that this results in a three times larger effort time wise for MFC to Qt ports in comparison to Qt3 to Qt4 ports for projects which are similar in size and complexity. Motif ports even result in a six times larger effort time wise compared to Qt3 to Qt4 ports.

The API of Qt is guaranteed to be stable between minor releases but not between major releases. A public example of the effort needed to port a code base to a new major version of Qt is the port of the KDE [1] code base from Qt version 3 to Qt version 4. At the start of the port the code base contained about 3.8 million lines of code, calculated using David Wheelers sloccount [13]. The KDE community started working on this port in early 2005, while the first non beta release of KDE, which is based on Qt4 was only in January 2008. Most of this work was done by community members in their spare time. However, several engineers currently working at KDAB were involved in this effort too. The knowledge gained and tools developed during this effort are still of great value during Qt-to-Qt porting projects.

We consider the Qt3 to Qt4 porting use case in this thesis to illustrate the improvements and solutions proposed. This use case is relevant in terms of the size of the API, size of code bases using the API, complexity of the code, and the type of users to be reasonably considered as illustrative for the challenges and advantages of the solutions presented in this thesis. Nevertheless, the solution presented here can be applied with limited modifications to porting other C++ software systems, or, in the presence of a suitable static analyzer, porting software written in other languages like Java.

In the next sections we will describe the estimation and porting processes, followed by the general requirements to which our work should comply. We will than have a closer look at the Qt3 to Qt4 porting process and describe the shortcomings with respect to the stated requirements.

1.1 MOTIVATION

Over time knowledge about porting projects and the risks of specific ports is built up. For KDAB, an important aspect is the ability to retain this knowledge and encode it a maintainable way.

Additionally, it should be stored in such a way which enables automation of the ports as much as possible. Currently, a large part of the knowledge is encoded in Perl[10] scripts and Emacs[4] macros. However, both resources become hard to maintain and use over time. A new way of encoding the knowledge is therefore required which can be used in tools that help with the estimation and automation of porting projects.

1.2 ESTIMATION

An important task at the start of a porting project is estimating the budget risks for the project. These risks can be in two areas. Firstly there might be code which is known to take a lot of time when being ported. Secondly there might be code which needs special knowledge which means that specific engineers have to be assigned to those parts. For an estimation of these risks various facts about the code base and the port are important:

- The source API - The API which is used in the code base before the port is performed.
- The target API - The API to which the code base has to be ported.
- API usage information - Detailed information about where in the code base the source API is used.

Porting is the transition of a code base using a source API to a code base using a target API. Detailed information about how the source API is used in the code to be ported is needed to make a correct estimation of the required effort and possible risks. Tools that understand the language can provide help in the following estimation tasks:

- Estimation of the overall complexity of the port.
- Estimation of the distribution of required changes over the source files of the code base.
- Estimation of the needed resources to be allocated for the project.
- Estimation of how to allocate resources needed for the project.

Currently scripts are run over the code base to get information about which parts of the source API are used. These are Perl scripts and have therefore no real understanding of the code. This information is therefore necessarily an approximation and partly unreliable.

1.3 PORTING

Porting a code base to a new API is a complex and time consuming process. The process consist of several steps and while performing these steps the engineers use a multitude of resources. Figure 1 gives an overview of the typical porting process. Boxes represent the various steps in the process and ellipses represent the resources used. In this section we first give an overview of the various steps in of the generic process and how the process influences and is influenced by the resources. Next we will apply this generic description to a specific instance of this process, namely Qt 3 to Qt4 porting.

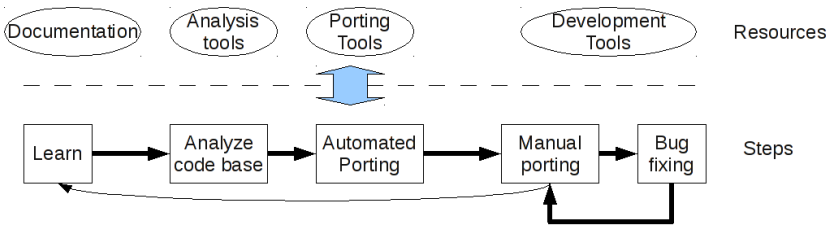


Figure 1.: General porting work flow.

1.3.1 *Learning*

A port starts by learning the APIs that are involved in the porting project. This involves gaining general knowledge such as the functionality the APIs offer and similarities between the APIs. More important however is in-depth knowledge about the APIs such as how particular constructs of an API should be used, the consequences of using these constructs in the wrong way and how to translate particular usage to the new API. The duration of this step typically reduces over time especially when the port involves an API which is well known by the engineers performing the port. Sources for learning are the documentation of the involved APIs and if available documentation specifically aimed at porting. Learning often results in additional documentation and tools for analysis and automated porting.

1.3.2 *Analysis of the code base*

Analysis of the code base serves two goals. Firstly, before a project starts analysis is done to estimate the duration and costs of the project. Secondly, during the port analysis is done to find recurring patterns in the code base. For these it can be more efficient to write new tools or adapt the current tools instead of doing the changes manually. The results of this step are typically new or adapted documentation, analysis tools and porting tools.

1.3.3 *Automated porting*

Porting a software system to a new API often has a sweeping impact on the system. The changes needed for a port may individually be easy. However, making all these changes by hand is very time consuming, boring and error prone work. Porting efforts can be described for a large part as a series of mechanical source transformations. Automated source code transformation systems offer a solution here. These systems can vary from simple grep like tools to systems which have a full understanding of the semantic complexities of the source language. The former are often easy to construct but limited in what they can achieve. The latter can be used for complex and precise transformations but are expensive to develop. However, the sheer number of changes needed for a typical port makes it economically viable to automate as much of the changes as possible, thus spend part of the budget on developing tools for automated transformations. When budget and resources are limited a trade off must be made between the level of correctness of these tools and the overhead of creating and using such tools.

Even when using state of the art source transformation systems, manual work is often required to finish the port. Manual porting starts with fixing compile errors resulting from the automated step. This includes adaptation of the build system to the new API, correcting parts of the code that were not transformed correctly and porting the parts of the code that were not changed by the transformation system.

In some cases, such as when porting a Qt3 code base to Qt4, there might be an intermediate API available for backwards compatibility. This reduces the initial amount of work needed to get rid of the dependency of the old API. When the initial port is finished, the port is completed by porting away from the intermediate API if needed and fixing known changes in run time behavior. During this step often new special cases are detected resulting in updated documentation and tools.

1.3.4 *Bug fixing*

Finally the project enters a loop in which the customer tests the ported software and reports bugs. Similar to manual porting, this step often leads to detection of new special cases. This step takes a large part of the projects time budget due to the fact that it can take quite some time when the problematic code paths are hit during run time. An additional problem is that these kind of run time errors are not always easy to reproduce.

1.3.5 *Requirements for a porting system*

Now we have outlined the porting process we can formulate the following general requirements (**GR**) for a porting system.

GR1. *Scalability*: The porting system should be able to handle real-world code bases containing millions of lines of code.

GR2. *Language*: A porting system requires a static analyzer to be able to reason about the code at hand and to provide the information needed for automatic porting. A static analyzer understanding only a limited subset of the language will thus definitely not work for large, complex, industrial code bases. Since the porting limitations caused by the limitations of such a static analyzer are very subtle and hard to grasp by actual programmers, we require our static analyzer used in porting to have a (nearly) complete understanding of the language, in our case C++.

GR3. *Simplicity of use*: The porting system proposed should be simple and quick to use by the typical engineer involved with the porting process. As such the solution should integrate tightly with the knowledge level and tool set used e.g. compiler, code editor and build system or IDE.

GR4. *Customizeability*: Different projects have different porting issues e.g. the types of constructs subject to porting, the actual porting rules and different styles of API usage. The users of the porting system should therefore be able to customize the system to cope with different porting scenarios.

GR5. *Predictable minimal impact*: Automation often results in a massive number of changes to the code base. It is near to impossible for an engineer to overview the actual impact and side effects of these changes. The changes made to a code base should therefore be minimal and have predictable effects.

GR6. *Transparency*: The porting solution proposed should allow developers to examine the impact caused by a porting action, or a set of actions, both before and after executing the porting. This is a crucial requirement when one assumes that the porting system in use does not work absolutely automatically and is not fully complete, that is, may need even the smallest amount of manual fixes to be done. Moreover, even if we assume a 'perfect' porting system which transforms the code always fully automatically from one working state to the next working state, users may want to see the impact that such a system will actually have on their code. This will help them to reason about various

related effects, such as build time increases, difficulties in learning the new code or changes to part of the code which are maintained by other developers.

In this thesis we are advocating a semi automated approach to C++ code porting as an efficient and effective solution for code porting tasks. For this solution to be efficient and effective, it should comply with the above mentioned requirements. In the following chapters, we will describe our proposal and also outline how this proposal complies with the above stated requirements.

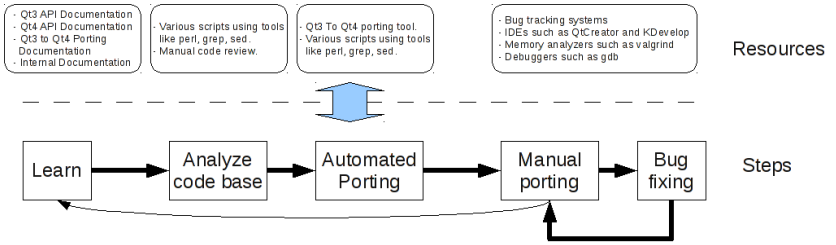


Figure 2.: The Qt3 to Qt4 porting work flow.

1.4 THE QT3 TO QT4 PORTING PROCESS

In this section we particularize the above discussion on code porting to the the specific Qt3 to Qt4 porting process. This process was the original motivation and carrier project for our work. Figure 2 shows the Qt3 to Qt4 instantiation of the general porting process depicted in Figure 1. The actual changes in the Qt4 API with respect to the Qt3 API vary from simple renaming of various language constructs to more complex changes such as changes in function signatures, change of semantic meaning of existing functions and replacement of constructs by completely new concepts. The more dangerous changes in the API are those due to which the run time behavior of the application changes. This can for example happen due to changed implementation of a function or due to changed meaning of function arguments. The problem with these kind of changes is twofold. The first problem is that during a port these changes do not expose themselves in the form of compile errors. Secondly, it is very hard to find the locations in the code where these particular problematic spots occur.

Practice has shown that performing the automated step in Qt3 to Qt4 porting process takes only about two percent of the time spent on the project while touching up to 80 percent of the code base in lines of code. Fixing the resulting errors is clearly less work than making all these changes by hand. Some of the resulting ninety eight percent of the time is spent on fixing build errors as result of the automated work. The larger part of it is

spent on fixing run time behavior regressions and new bugs introduced during the porting work.

We next give a brief overview of the current state-of-the-art used in Qt3 to Qt4 porting at KDAB and highlight the limitations of this solution with respect to the general requirements stated in Section 1.3.5.

1.4.1 *Scripts*

For estimation as well as for automated porting the current process heavily relies on scripts. These are mainly perl scripts[1] which make use of regular expression to find various language constructs. Although the approach is not purely based on regular expressions and in some cases not even context-free, it is clear that the scripts do not have a real understanding of the semantics of C++. This therefore clearly violates **GR2**.

Another problem is that different engineers tend to have slightly different approaches in solving similar problems when using scripts. This results in a collection of scripts which becomes hard to use and maintain over time which is a violation of **GR3**.

Scripts also come with the problem that one cannot know beforehand which parts of the sources will be touched by scripts. This may be actual code but even likely code that is commented out or just plain comments. Because of this it is hard to tell what the actual side effects of the changes made by scripts will be and therefore this approach also violates **GR5**.

Finally, scripts by themselves provide no transparency at all. At best they can output data in a specified format which can be used for further processing. Hence, this approach violates **GR6**.

1.4.2 *The qt3toqt4 porting tool*

The Qt4 framework comes with a tool which tries to automate the most tedious part of the porting effort. It reads an XML file containing porting rules. These rules can be used to rename classes, prefix or rename enumeration values and add or modify include directives. Although this tool does make use of a C++ parser, it seems to do only a partly semantic analysis. This results in incorrect transformations for simple cases such as renaming of enumeration values. More advanced cases such as changed signatures of functions are not handled. This limited understanding of the language is a violation of **GR2**. Moreover, just renaming constructs may lead to unpredictable run time behavior and therefore this approach violates **GR5** as well.

1 The scripts can be found in the KDE svn repository: http://websvn.kde.org/trunk/KDE/kdesdk/scripts/qt4/

1.4.3 *IDEs*

For manual development editors, ranging from plain text editors to complete IDEs are used. IDEs such as KDevelop [7] and QtCreator [12] become more and more powerful nowadays. They offer features like integration of the build system with the editor to allow the engineer to quickly jump to the location in the source code that causes the compilation error. IDEs also help the engineer in understanding the code by providing context aware highlighting and links between code and documentation. Some IDEs even provide refactoring support [2] [14] but for C++ this is in most cases limited and constrained to a fixed set of refactorings. Currently we do not know of an IDE which directly tries to meet our stated requirements. However, an IDE with good understanding of the semantics of C++ would enable the development of a framework that meets the stated requirements.

1.5 PROPOSED SOLUTION

In this thesis we present a tool-based approach to improve the efficiency and the effectiveness of estimating the effort and performing the actual port of a C++ code base from Qt3 to Qt4. To this extent we integrated automated analysis and transformation capabilities into an IDE. These capabilities are supported by additional visualizations which support the user on the task of estimation as well as with the actual porting. Our solution tries to meet the requirements as stated in Section 1.3.5.

We implemented a prototype as a plugin for the KDevelop [7] IDE. Making use of its enabling technologies like parsing and semantic analysis of C++ code and editor integration. The plugin consist of three components. The first component is a query engine which uses the results from the parsing and semantic analysis to find affected code constructs in the analyzed project. It is specifically aimed at finding uses of a specified API. The second component is a transformation engine. This engine takes results from the query engine and transformation descriptions to apply changes to the source files. The last component provides two visualizations. One visualization gives an overview of the project that has to be ported and is aimed at helping the engineer to estimate the effort and difficulty of a porting project. The second visualization is aimed at guiding the actual porting process by giving an overview of where changes will be applied by the porting engine in a particular file.

1.6 ROAD MAP

This thesis is structured as follows. Chapter 2 gives a short introduction on the topic of fact extraction from C++ code bases, next the requirements for a fact extraction framework suitable for porting will be discussed. Finally, a review based on these requirements of various fact extractors will be done.

Chapter 3 discusses the requirements for a C++ code transformation system, followed by a review of various C++ transformation systems. It concludes with a brief discussion on fact extraction and transformation support of C++ IDEs.

Chapter 4 gives an overview of the architecture of our porting framework. In addition it discusses some of the internals of the KDevelop IDE which are of importance for our query and transformation engine.

Making changes in a reliable way to code bases requires an engine that can find the code that is subject to change in a precise way. Chapter 5 discusses the design of our light weight query engine, built as extension of the KDevelop C++ analyzer.

Chapter 6 discusses the design of our transformation engine.

Chapter 7 illustrates the use of both the query and the transformation engine. We do this by taking various transformation use cases from the Qt3 to Qt4 porting process.

Chapter 8 presents and discusses the visualization techniques, added to our plugin to support various use cases related to porting.

Finally, Chapter 9 reflects on the previous chapters and discusses to what extent we were able to achieve the goals we stated in the introduction.

Part i.

Related Work

2 FACT EXTRACTION

Compiler technology is a set of techniques and tool implementations which deal with the processing of source code for different purposes. The most well known application of compiler technology is probably in compilers, generating executable programs out of source code. However, as pointed out in Aho et al. [15], there are many more applications of compiler technology, the most important in our context being static analysis and program translations. In this chapter and in the next chapter we will have a closer look at compiler techniques and integration with IDEs and argue why we chose specific techniques and tools.

Program translations are normally thought of as translations of a high level language to machine language. However, the same technique can be used for transformations between two different high level languages or for transformations within the same language. This topic is discussed in more detail in Chapter 3.

Static analysis serves the extraction of various facts from a code base without actually executing the code. These facts in turn are used for various tasks related to the code at hand. They can be used for example to assess the quality of the code, to construct simplified visual representation which support tasks such as program comprehension or to perform transformations on the code. The facts delivered by static analysis are quite broad, ranging from basic raw facts to more refined facts. Raw facts include lexical information of the source files, abstract syntax trees which describes purely the syntax of the code and abstract syntax graphs which describe the syntax and semantics of the code with respect to the rules of a given programming language. The more refined facts include quality metrics such as cohesion, coupling and complexity but also design patterns, architectural patterns, call graphs, inheritance graphs and control graphs. An extensive overview of quality metrics and their use can be found in Lanza et al. [26].

2.1 FACT EXTRACTION FROM C++ CODE BASES

Given our requirements (Section 1.3.5), we need to perform static analysis of large, complex C++ code bases. The analysis should deliver us enough facts, to be able to efficiently and effectively implement our required program transformation goals for code refactoring and porting. Given the open scope of our automated porting challenge, we have to be able to define porting rules on a

wide range of C++ constructs. Moreover, the porting rules need to have full access to lexical, syntactic and semantic information on the code to be ported. Hence, we need a C++ analyzer which is able to efficiently and effectively provide such information[1]. In the next section we will first discuss the specific requirements for a C++ fact extraction framework. We will use these requirements to evaluate various fact extraction frameworks.

2.2 REQUIREMENTS

Boerboom and Janssen [20] give an overview of the general requirements for C++ fact extraction frameworks. In this section we reiterate over these requirements and adjust them too the specific needs of this project where needed.

FX1. *Fault Tolerance.* Fact extraction from code that that does not compile must be possible. Porting projects can happen on code bases that require a complex build configuration which might not always be completely reproducible when working off-site. Also, the code might become partly invalid when changes are applied to individual files as result of transforms or manual editing during the porting process.

FX2. *Completeness and correctness.* All parseable, syntactical constructs should be extracted correctly. This is especially important in case of porting because the Qt API can be used anywhere in the code base.

FX3. *Compliance.* The parser should at least understand the C++ standard which is required by the Qt3 API. However special knowledge of different dialects like g++, Borland C++ and Visual C++ is not of particular importance. Lack of such knowledge should have a minimal impact on finding facts of interests (i.e. use of Qt3 API). This is due to the fact that most software using Qt, was based on Qt to make it cross platform. Meaning, the code is often compiled with various compilers and therefore contains very little compiler specific code. In addition, Qt contains many convenience classes which hide platform specific issues from the engineers.

FX4. *Cross References.* Cross references in the source should be resolved correctly and complete enough to support the task

1 We make a clear distinction between a parser and an analyzer. Whereas a parser delivers AST information, which is in principle sufficient to determine the nesting of syntactic constructs in the source code, a semantic analyzer goes much beyond that. An analyzer delivers relations between code elements such as dependencies, uses, inheritance, scoping, type compatibility, type subsuming and type equivalence between classes. All this information is essential for implementing a truly effective system for code transformations, which is our aim.

of finding and porting Qt3 API. This means that uses of classes, class member functions and public class and namespace members are resolved by the framework. Because the Qt3 API makes use of templates, though in general not in very complex way, there should be at least rudimentary support for templates too.

FX5. *Preprocessing.* The preprocessor should be able to process source code of arbitrary complexity. It is especially important that line and column information of tokens in the original source file are kept and available for later use. The preprocessor should therefor deal correctly with macro expansion. Especially in this context, because we want to touch as little code as possible when performing transformations.

FX6. *Coverage.* Because of the fact that the C++ grammar is not fully context independent, possible ambiguities may occur. The framework should be able to deal with this to a reasonable extent. The main area of interest is Qt3 API and the experience is that such areas need extra work either during or after the initial port anyway.

FX7. *Output completeness.* The output of the framework should be complete with respect to the correctly parsed and analyzed input. This includes but is not limited to line and column information of syntax constructs and type system information. If an analyzers output is severely limited, then its use as a building block in a program transformation framework will be limited to the type and extent of information that it provides to the designers of the transformation framework.

FX8. *Performance and Scalability.* The time to extract facts from code bases should be similar to compilation times of the same code. In addition, support for incremental updating of a particular source file is preferred as source files will change often during the port.

FX9. *Portability.* The framework should be available on the major platforms used by the principal, which are Linux and Windows.

FX10. *Availability.* The framework should be available in one of the open source variants. Which means that at least for the prototype commercial products are not a candidate.

In addition to these requirements there are some more specific requirements for this project.

FX11. *Integration with IDE.* Because our usability requirement **GR3** the framework should integrate well with an IDE. This

makes sure that the automated porting tasks integrate nicely into the work flow of the developer during the porting project. In Section 3.6 we will discuss the IDEs that we evaluated for our purposes.

FX12. *Ease of use - Build system support.* The framework should not require a lot of effort to set up. Build information needed to parse the files of a project should preferably be extracted from the build system of the project.

FX13. *Generality - Build system support.* As the expected result is a prototype, it is not needed that it has support for all possible build systems. The prototype should at least be able to handle Makefile managed and CMake managed projects.

FX14. *Coverage - Query engine.* The query engine is not required to return results which are in parts of the source files that are disabled due to preprocessor definitions.

FX15. *Additional Qt language extension support.* The framework should understand the Qt Signals and Slots mechanism. The signals and slots mechanism is one of the core technologies of Qt used for communication between objects. It is a runtime mechanism, meaning that normal compilers cannot give useful warnings when using this technology in the wrong way. We therefore give a short overview of how the mechanism works.

Signals are emitted after particular events and other objects can connect one or more slots to a signal. A slot is called as soon as the signal to which it is connected is emitted. To understand the signals and slot mechanism, the preprocessor must be extended because the signal and slot mechanism is implemented by means of special macros. However, some of these macros actually do not expand to real C++ syntax but are used as a markers.

Listing 1 shows a slightly simplified use of the signal and slot mechanism as provided by Qt. In a normal build process a separate tool, called the meta object compiler (moc) is run before the actual compilation of a file. This tool parses the file and looks for the signal and slot mechanism macros (i.e. Q_OBJECT, slots, signals, SIGNAL, SLOT). It than generates a header which should be included by the processed file.

The SIGNAL and SLOT macros expand their argument to character array. So after preprocessing, without storing special knowledge it is not possible to say where these macros

were used. This information is important for checking signals and slot connection correctness and also for renaming signals and slots.

Listing 1: A simple signal/slot example

```
class MyObject : public QObject
{
  Q_OBJECT

public slots:
  void slotA(int a);

protected slots:
  void slotB(const MyObject &);

signals:
  void mySignal(int a);
};

...
  // Connect is a method of QObject.
  connect(someObject, SIGNAL(mySignal(int)),
          this, SLOT(slotA(int)));
...
```

In the next sections we will review various C++ fact extraction frameworks with a focus on the requirements discussed in Section 2.2. We will briefly describe the used techniques and applications of these frameworks. We do not consider compilers which support extensions such as GCC 4.4 because these frameworks fail to meet requirements **FX1** and **FX5**. Where applicable we will also clearly indicate when a given framework fails to meet one of our requirements.

2.3 ASPECTC++

AspectC++ Spinczyk et al. [32] is an extension of the C++ language to implement support for Aspect Oriented Programming (AOP) with C++. This extension consist of about ten grammar rules in addition to the original C++ grammar as presented in Stroustrup [33]. These additional rules introduce the AOP concepts like so called "point cuts", "advices" and aspects. These language constructs can be used by the programmer as if they where part of the language and a special tool called the aspect weaver inserts code fragments at the appropriate locations at compile time. The generated code is passed to the real compiler.

The aspect weaver is based on a fact extraction framework called PUMA [31]. This framework contains the whole set of

tools needed for fact extraction. It provides a lexical scanner, an integrated preprocessor, a parser and a semantic analyzer.

Although the framework seems to support a reasonable subset of C++, it does not support C++ templates according to the aspectc.org website which is a violation of requirement **FX4**. Another drawback is that no active development seem to have recently happened on this project. The latest release of the complete AspectC++ framework dates from 2006.

2.4 COLUMBUS CAN

Columbus Ferenc et al. [24] is a reverse engineering framework for C++ projects. It is designed to analyze large C++ software systems and presents the information in a common specification called Columbus Schema for C++. Projects are processed in three stages. First an extractor parses the source files of the project. Next a custom linker is used to extract information with respect to the modularity of the project. Finally the gathered information is passed to an exporter which has plugins to export the information into different kinds of formats. In addition to these steps the framework also offers filtering of the information to reduce the size of the output information.

The actual extraction from source files is done by two tools called C++ Analyzer (CAN) and C++ Analyzer Preprocessor (CANPP) Arp et al. [18]. These tools are called during the normal build process by means of compiler wrapping. Meaning that instead of the actual compiler, these tools are called which than forward the command to the real compiler. This ensures easy integration with existing projects. The parser meets the ISO/IEC standard of 1998 and the grammar has been extended to support various commonly used dialects.

Large projects can be parsed with the Columbus framework, which suggests a high quality parser. It also performs a semantic analysis, however the parts of the Annotated Syntax Graph (ASG) for statements and expressions is not created. This will make it hard to find uses of symbols and information needed for refactoring and therefore requirement **FX4** is not met. Furthermore the schema does not store lexical information which violates requirement **FX7**. Finally, the extracted information is not directly accessible via an API. Meaning, that extracting information would need an engine which understands one of the output formats and on top of that an engine which enables searching for the required information.

2.5 DMS

DMS Baxter et al. [19] is a commercial program analysis and transformation system from Semantic Design. It is designed for analysis and transformations on complete systems. DMS is therefore able to not only handle C++ code but also other widely used languages (or domains) in industry such as Java and COBOL. One of the foundations with respect to fact extracting are the so called hyper graphs, a generalization of graphs. This representation makes it possible to capture any arbitrary graph like language. AST are encoded on top of the hyper graph structure.

Parsers for a domain are generated using a domain specification and the present parsers are implemented with support for integrated lexing, preprocessing and parsing. An interesting feature is the general strategy to avoid macro expansion. The thought behind this is that DMS should process what the programmer sees. DMS uses GLR parsing which eases the detection of ambiguities that may arise due to the nature of the C++ grammar.

DMS also provides a general symbol table management system. This system is used for storing and providing access to name and type information and their associated symbol spaces (i.e. namespaces in C++). On top of this generic symbol system a domain specific API is constructed which enables looking up symbols in a specific context. The C++ name and type resolver is extended to have support for preprocessor directives Akers et al. [16]. This means that name lookup can be done for any context in which the code is used.

DMS is a state-of-the-art tool for static analysis and has been used in numerous industrial projects for program analysis and transformation, including C++ programs. It will however never be an open system in terms of API and therefore fails to meet the availability requirement **FX10**. Also, being a fact extraction and fully automated transformation engine might make it harder to integrate it with a rapid development process. Furthermore it is highly likely that given the complexity of the system it has a steep learning curve and cannot be easily integrated into the current work flow of the developers and therefore violates **GR3**.

2.6 KDEVELOP

KDevelop [7] is a powerful IDE built upon the KDE platform. The latest stable version is available for all major platforms. KDevelop is based on a set of libraries which implement functionality for IDE like programs called KDevPlatform. Among the functionality in KDevPlatform is project management and programming language independent language support. The language

support offers features like a generic ASG, called the Definition Use Chain (DUChain) internally, editor integration and background parsing. The language support is highly adapted to rapidly changing documents.

KDevelop integrates these functionalities in one IDE by means of plugins and implements on top of the language library support for PHP and C++. The C++ fact extractor makes use of an internal preprocessor which retrieves the environment information from the project management module. As a natural side effect of being based on KDE platform and therefore based on the Qt toolkit too, the C++ fact extractor can deal very well with Qt based C++ source code. The preprocessor is adapted to not expand signals and slots macros. These are kept in the preprocessed source and the parser is adapted to handle them. This is a similar technique as described in [19]. The parser is able to successfully parse large and complex code bases in a fault tolerant way. The latter is especially important to support incremental updating of extracted facts during editing of documents. Due to the integration with the editor, low level information like positions of tokens and macro related information is stored too. The semantic analysis is rather complete. It has substantial support for the more complex steps like template instantiation and function overload resolution. Extracted information is stored in a repository and is accessible via API.

KDevelop is available under the GPL license and there is a LGPL library included which enables extension of KDevelop by either open source or proprietary plugins. The API for querying the extracted facts is aimed at use through an editor. This can be overcome relatively easy by extending the API to meet our needs. We did not find obvious limitations with respect to the stated requirements for fact extraction.

2.7 SOLIDFX

SolidFX Telea and Byelas [35] is an integrated environment for industrial code analysis. It is based on the fact extractor presented in Boerboom and Janssen [20]. This is a tolerant, heavyweight extractor. It is tolerant in the sense that it will try to recover from lexical errors as much as possible. In contrast to lightweight parsers, which do only partial parsing and type checking of the source code, the SolidFX fact extractor does preprocessing, parsing and full semantic analysis of the source files. All information extracted from a translation unit is filtered for unneeded symbols and than stored in a so called fact database. On top of the fact database a query engine is built which enables querying the extracted facts by means of different kinds of queries like AST visitor queries, preprocessor queries, type queries and location

queries. Queries can be composed into query trees which make it possible to extract complex information from any fact database. The queries can be described in XML files which are read by the engine. It is also possible to use the C++ API to query the extracted facts.

The above described properties make the fact extractor of SolidFX a good candidate for our work. However, the framework was built with fact extraction and visualization as major goal. It therefore has no integration with an IDE. Also, it is meant to be a general fact extractor, so we expect no special knowledge available with respect to the signals and slot mechanism provided by Qt.

In this chapter we discussed the topic of C++ fact extraction. We enumerated the requirements for a fact extractor that is suitable for use in an automated porting system. These requirements were used to assess various freely available and commercial fact extractors. From this assessment we conclude that the KDevelop fact extractor is a suitable choice for our further work.

3 TRANSFORMATION SYSTEMS

After fact extraction (Chapter 2), the facts are available in some intermediate representation which can be used for further processing. In our case we will use these facts in combination with transformation descriptions to apply transformations to the code in a semi automated way.

In the literature we find several forms of program transformations. Procedural transformations, being arbitrary functions applied on AST's are, most commonly known for their use in compilers. Software refactoring frameworks are starting to use this technique too. In this context we briefly discuss the Stratego/XT (Section 3.3) and the Transformers (Section 3.4) frameworks. Both frameworks use the procedural transformation technique and are used in the context of transforming C++ code bases, which is our area of interest.

Another form of transformations found in the literature is the so called source-to-source transformation, being a mapping between the concrete syntax forms of the code. In its simplest form this is a simple search and replace as using grep like tools. A more advanced framework which enables source-to-source transformations is the ASF+SDF (Section 3.2) framework. Finally, there is also a hybrid solution described in DMS (Section 3.5) which combines compiler techniques with source-to-source transformation.

The form of transformation is important with respect to the resulting code one can expect from the framework. As most of the porting projects at KDAB are done on code bases which are intended too be human readable it is important that all syntactical information is kept intact as much as possible.

In the following sections we will first outline the requirements for our transformation engine. Next we will describe various transformation systems. Some of them, such as Transformers, are specifically created for C++ source code, others are general transformation systems which are described to illustrate the techniques used for transformations.

3.1 REQUIREMENTS

As with the fact extractor, the transformation engine has to adhere to some requirements in the context of our problem. In this section we describe the requirements for the transformation engine.

TF1. *Preprocessor.* All preprocessor information should be kept intact when it is not part of code that is affected by a transformation.

TF2. *Source code layout.* The code layout should be kept intact for all code that is not affected by a transformation.

TF3. *Transformation correctness.* The framework is aimed at supporting engineers with experience in porting software to Qt4, therefore there will be no support for checking correctness of defined transformations.

TF4. *Minimal impact.* No changes, other than specified by the transformations should be made on the code.

In the following, we briefly review existing program transformation tools with a focus on C++ and outline their advantages and limitations with respect to the above listed requirements.

3.2 ASF+SDF

The ASF+SDF Deursen et al. [21] framework is a generic framework to define languages and generate tools for these languages. The formalism allows the specification of arbitrary syntax using the Syntaxt Definition Formalism (SDF). From these definitions parsers are generated which use Generalized Left-to-right Rightmost (GLR) parsing techniques for the defined languages. Additionally, tools such as type checkers and pretty printers can be generated for the specified languages. The system also supports transformations which are described in terms of concrete syntax. The fact that the framework is language independent enables an approach similar to DMS (Section 3.5). Using the pure ASF+SDF framework would require a lot more work than can be done within the time frame of this project. The major problem is that ASF+SDF does not integrate a semantic system, but just a syntactical one. Moreover, it does not integrate a C++ front-end with semantic information. Hence, while ASF+SDF may be an interesting academic development, it is not applicable to analysis and transformations of large and complex C++ code bases.

3.3 STRATEGO/XT

Stratego/XT Visser [38] is a framework to develop transformation tools. It aims to have support for a wide range of program transformations. The framework consists of two main components. The first component being Stratego, a language to describe the transformations. The language consists of transformation rules and a transformation strategy. Unlike ASF+SDF, the Stratego

transformation rules describe basic transformation steps on an AST in stead of on concrete syntax. The rules are combined in a transformation strategy to form a complete transformation. The second component of Stratego/XT is XT, a set of tools providing facilities for the infrastructure needed for transformation systems. Besides parsing and pretty printing tools it also contains a transformation tool and a transformation system. The transformation tool is a wrapper around a set of transformation rules and strategies which can be called from the command-line. A transformation system is in turn a composition of these tools together with other facilities like a parser and a pretty printer, able to perform a complete transformation. Like ASF+SDF (Section 3.2), Stratego/XT uses SDF for specifying language syntax and generating parsers. Being based upon ASF+SDF, Stratego/XT suffers from the same limitations.

3.4 TRANSFORMERS

Transformers Anisko et al. [17] is a transformation framework specifically for C++. The main motivation for transformers is to simplify generic programming. This is done by means of transforming code written in the usual C++ style into code that more extensively uses the generic programming concepts. It uses a parser generated with ASF+SDF framework and the Stratego language for specifying the transformations. Although promising due to the tools it is based upon, Transformers has some limitations which make it unsuitable for our goals. First of all, the parser for the C++ language is not yet complete, especially the missing support for template based constructs makes it unsuitable. Furthermore, Transformers works on preprocessed documents. This results in code that is pretty printed in a way that might be completely different from the original code and has lost all comments from the original code too. Even worse, the code will be polluted by everything that the preprocessor has pulled in.

3.5 DMS

In Section 2.5 we already gave an overview of the fact extraction part of DMS. The DMS system also incorporates a transformation engine Baxter et al. [19]. This engine offers interfaces for procedural manipulation of general hyper graphs and ASTs. It also has an AST-to-AST rewriting engine. Besides those procedural manipulation methods it is also possible to define source-to-source transformations based on the language syntax. DMS is designed to cope with systems with tens of thousands of source files and millions of lines of code. For this reason a parallel language, called PARLANSE, was developed to support parallelism for

symbolic manipulation. One of the unique features of the DMS system is that it is possible to mix procedural transformations with source-to-source transformations. An example of this is using so called attribute evaluators to verify the applicability of a transformation at a specific point in the AST. Besides extensive APIs to implement transformations there is also the possibility to specify the transformations in the domain notation of interest. This saves the engineer the burden of getting known with the details of the language at tree representation level.

3.6 SOFTWARE UNDERSTANDING AND REFACTORING SUPPORT IN IDES

As outlined in Chapter 1, one of our main requirements is to have a program transformation solution for porting code which is easy to use. Our targeted developers mainly work with IDEs, which is also the case in our carrier project at KDAB. Hence, getting insight in the state-of-the-art support of IDEs for program understanding and refactoring, both being requirements for code transformation, is necessary. The following sections provide this insight with a focus on several mainstream IDEs.

Various IDEs supporting C++ development exist nowadays. In this section we give an overview of some freely available IDEs which we have found to be able to at least partly support our needs. The IDEs we tested are Eclipse [3], QtCreator [12] and KDevelop [7]. Eclipse started as a Java IDE but gained advanced C++ support due to the CDT plugin [2]. It was chosen because it has some advanced refactoring features for C++ which we did not find in other IDEs. QtCreator and KDevelop where chosen for their close relation to and good support for Qt based projects. We will now briefly describe the offered functionality available in these IDEs with respect to program understanding, refactoring and querying.

3.6.1 *Program understanding*

All three IDEs support the task of program understanding by making the code discoverable using several techniques. The first technique we found in all IDEs is context aware highlighting. Although the implementations vary slightly between the IDEs it is clear that the code highlighting is not just simple highlighting by keyword but makes use of semantic knowledge about the code. E.g. type names have different colors than variable names, local variables different colors than global variables, etc. A second technique is a small pop up window which appears when hovering over a symbol. This window shows at least the definition of the symbol and when available documentation or a link which opens

the documentation in another part of the IDE. Another technique used is the so called "follow symbol under cursor". This enables the engineer to switch between definition and declaration of a symbol by clicking the symbol in the editor while simultaneously pressing a modifier key such as ctrl. QtCreator in addition provides a dense pixel technique for understanding the structure of a source file similar to the shaded cushions as described in Lommerse et al. [27]. The QtCreator implementation does not make use of cushions but uses different shades of gray to give a hint about the level of nesting.

3.6.2 *Refactoring*

The only kind of refactoring supported by all three IDEs is renaming of symbols. Eclipse offers the possibility to rename occurrences in comments and macros too, while the other IDEs only rename symbols in code. All of them show the impact of the rename action in a similar way at first. A tree like structure is shown containing files and affected lines. Eclipse and QtCreator offer the possibility to exclude lines from the rename action. Eclipse in addition has a preview step in which a diff viewer is used to show the differences between the affected files before and after the action. The diff viewer shows only one file at a time. Both KDevelop and Eclipse support creating the implementation of a declared method. Eclipse in additions supports slightly more advanced refactoring actions such as generation of get and set methods for class members, extracting part of a function body into a separate function, extracting constants and extracting local variables. The approach to present the impact of these refactoring actions is similar to the approach for presenting the impact of renaming a symbol.

3.6.3 *Querying*

None of the IDEs offer querying of the code base as a separate function. However, all of them support the so called "uses" functionality. This enables the user to find the uses of a symbol such as a function or a class. Eclipse is the only IDE which offers the possibility to reduce the scope of the results to include only results in a selected project. In all three IDEs only one symbol at a time can be queried for uses and the results are presented in a tree widget.

In this chapter we discussed the topic of program transformation. First, we enumerated the requirements for a transformation engine that will become part of our porting system. These requirements where used to assess various approaches to program

transformation found in the literature. The approaches can be devided in procedural transformations, i.e. functions operating on an AST and source-to-source transformations, i.e. transformations based on the language syntax. Given the problems with most of the procedural approaches we decided to use source-to-source transformations but enhance these by making use of the semantic knowledge provided by the fact extractor. We also discussed the current state of the art for querying and transformation support in modern IDEs and pointed out several areas that need improvement with respect to our use case.

Part ii.

Design and implementation

4 FRAMEWORK DESIGN

In this chapter we describe the design of our semi automated porting framework for C++ code bases. To ensure tight integration with the normal development process we based our solution on the KDevPlatform framework and implemented it as a plugin for the KDevelop IDE. KDevelop was chosen because of its robust C++ parsing and analysis framework. This framework is not only able to parse and analyze large and complex code bases, but also can handle code containing errors very well. Therefore the choice for KDevelop covers most of the requirements for the fact extractor as stated in Section 2.2. In addition, KDevPlatform offers facilities needed in our context such as editor integration and project management. These properties together form a firm foundation to build a framework that meets the requirements we stated in Section 1.3.5. Finally, some of the engineers at KDAB also use KDevelop as their primary IDE, meaning that a plugin for KDevelop will seamless integrate in their daily work flow. A more detailed overview of KDevelop and how its facilities are used will follow.

The aim of our framework is twofold. Firstly, it should offer a query functionality and visualizations that helps estimating the effort of a porting project. Secondly, we want a framework that transforms code that is queried for. An engineer thus needs to be able to specify which API is subject to transformation. We therefore need a framework that is able to read query specifications and use that to query a code base. In addition it must enable the engineer to specify transformations that can be performed on locations in the source code found by the query mechanism.

In this chapter, we describe the global architecture of our framework. We will also detail the way in which we used and extended the static analysis capabilities of the KDevelop IDE which we chose as a basis to build upon. The query engine used to identify source code fragments subject to transformation is outlined in Chapter 5. The actual code transformation engine is described in Chapter 6.

4.1 ARCHITECTURE

Figure 3 shows an overview of the architecture for our framework. The yellow colored components are coming from the KDevelop framework. The fact extractor resembles the typical compiler pipeline and will be detailed in Section 4.2.2.

Query and transformation descriptions are stored in XML files which are read by the framework. These can be edited and reloaded while the framework is running. This ensures a low usage overhead and addresses **GR3** as well as **GR4**.

The query engine performs the queries by using the symbol table and the AST representation of the source code. Results of a query execution, so called hits, are passed on to the transformation engine. This engine uses the properties of a hit and a transformation description to perform transformations. These are performed directly on source code. The results of the query engine can in addition be passed to two different views we designed to support the estimation and porting tasks in a visual way.

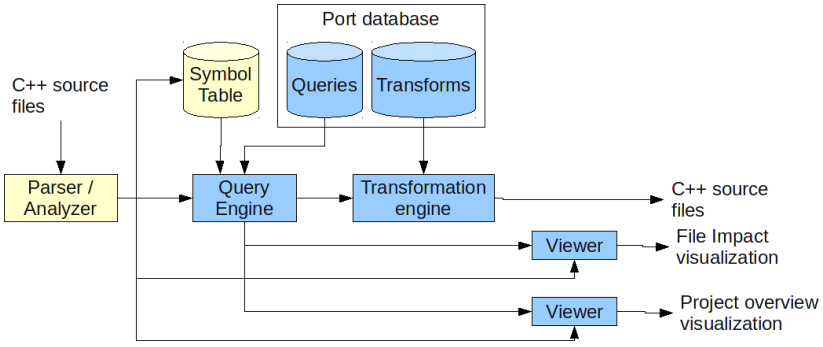


Figure 3.: Architectural overview

Our transformation system works in three phases:

1. The parsing and analysis framework of KDevelop processes a source file.

2. The resulting information is passed onto the query engine. Query specifications from an XML file are read by the query engine and it performs the specified queries.

3. Either the user examines the results of the queries for a complete project or he examines the results for a specific file and performs the available transformations.

These phases and visualizations will be detailed in the next sections and chapters. In Section 4.2 we will describe the KDevelop components that are involved with project handling, parsing and analysis of source code as these components cover some of our stated general as well as fact extraction requirements. Next we will describe the query engine in more detail in Chapter 5, followed by a detailed description of the transformation engine in Chapter 6. Finally, we will discuss the visualizations we added to support estimating and porting in Chapter 8.

4.2 KDEVELOP

KDevelop is a means to meet some of our requirements in the first place. However, it also influenced the design of our query engine and transformation engine due to its internals. In the next sections we give a brief overview of KDevelop and which requirements are met by the functionality it offers. We will especially give an overview of its parsing and analyzing capabilities.

KDevelop itself is a powerful IDE that is built out of several components which are provided by KDevPlaform. Both KDevelop and KDevPlatform are based on the KDE platform which in turn is based on Qt. The major use case supported by the KDevelop IDE is Qt and KDE software development. This was a main consideration for choosing this IDE as foundation for our porting framework. Being based on Qt/KDE means that it makes use of API which already known and therefore it is easier to extend and maintain if necessary. It was also expected that KDevelop has additional support for Qt specific C++ use such as the signal and slot mechanism which we briefly described as part of requirement **FX15**.

KDevPlatform is a bundled set of components which provide the functionality which one needs in general to build an IDE. They are built up as generic components providing interfaces which can be implemented for specific needs. For example, there is a generic project manager and there are implementations for Makefile based projects and CMake based projects.

4.2.1 *Project handling*

Fact extractors, or static analyzers, in general operate on single translation units, i.e. source files and subsequently included header files. Information required to perform a correct parse and analysis of a source file such as include paths and defined preprocessor macros are passed to the extractor in various ways such as via command line or configuration files. Extracting this kind of information for each file in large projects is a cumbersome task which should be dealt with automatically in line with our scalability requirement **GR1** and our usability requirement **GR3**.

KDevPlatform provides various interfaces and plugins which make it easy to deal with this problem. First of all there is the `IProject` interface, which defines the project concept for KDevelop. This interface can be used to retrieve all files of a project. Next there is the `ILanguageController` interface. KDevelop offers among other languages support for the C++ language. This interface deals with getting the right language support implementation for a given language. The interface also provides the possibility to determine the language of a given file. Together

with the IProject interface a simple filter can be build which retrieves all C++ files from a given project.

Each `IProject` in KDevelop also has a so called `IBuildSystemManager`. This interface deals with build related topics such as the build directory, include directories, defines and targets. There are two implementations of this interface, one to manage CMake projects and one to manage plain Makefile projects. Both builders parse the relevant files of the project (e.g. CMakeLists.txt files for CMake based projects) and extract the information needed to parse the files correctly. This all happens automatically when opening a project in KDevelop and therefore ensures seamless integration with the normal work flow of an engineer.

4.2.2 *Fact extraction*

The KDevelop C++ fact extractor resembles the typical compiler pipeline. It consists of a custom preprocessor, a parser and an analyzer, the so called DUChain builder. The whole parsing stack is wrapped in the `BackgroundParser`, which is a multithreaded parse job scheduler. Parse jobs can be started simply by passing the URL of a source file to the scheduler. The parsing is performed asynchronously and a callback function is called when results are available. The fact extracting stack has error recovery methods implemented at various levels to deal with invalid code and other things that might go wrong during parsing such as missing include paths or invalid syntax constructs (req. **FX1**).

PREPROCESSING For our transformation engine we need a preprocessor that is able to deal with arbitrary complex code, moreover it should correctly report token locations. The preprocessor of KDevelop is a custom written preprocessor which creates a token stream of a source file and its includes. The exact token positions are stored for later use (req. **FX5**). Tokens that are the result of macro expansion get an invalid location assigned. An interesting feature of this preprocessor is that it puts custom tokens in the stream for the Qt signal and slot macros (req. **FX15**).

PARSING The parser is a hand written LL(k) parser which can parse practically all C++ syntax constructs (req. **FX6**). It constructs the AST which consist of about 80 different node types. A default visitor class for the AST is provided and can be subclassed for custom actions on the AST such as building the type system. The number of AST node types is quite low in comparison to other fact extractors such as the ELSA based extractors. Boerboom and Janssen [20] report 300 node types in the ASG. One reason for the

difference between the number of node types is that semantic type nodes in KDevelop are stored in the DUChain and are not combined in one ASG as done in ELSA based parsers.

Another, reason is the fact that the AST of the KDevelop parser is highly simplified, meaning that syntactically similar, but semantically differing code constructs get the same AST representation. Other parsers such as the Eclipse CDT C++ parser and the QtDesigner C++ parser use a similar lightweight AST description. This simplifies the parser design, as fine-grained distinctions between syntactically similar, but semantically different constructs do not have to be made at parse time. Such distinctions are sometimes hard to make for C++ in the parse phase and without a full semantic analysis. For example, without semantic analysis it is hard to make a distinction between C-style casts and normal function calls. We discovered that this actually is a disadvantage because one ends up with a lot of special cases when visiting a particular node of the AST looking for specific information.

ANALYZING KDevPlatform has a so call DUChain [1], which is a language independent representation of source code. The DUChain is built up in two phases. In the first phase a visitor is used to construct a sequence of scopes, so called `DUContext`, in a source file. Each context has a parent context except the `TopDUContext` of a file, which represents the global scope. For each context the associated definitions are stored too in the first phase. In the second phase the uses of declarations are constructed. A use in KDevPlatform is actually the combination of a declaration and a range in a document where the declaration is used. This is quite a different approach than we see in other frameworks such as ELSA, where the complete representation is stored in one ASG. Some convenient API to get from a specific range to an use exists though. With respect to the resulting cross references we did not find limitations in the context of Qt3 to Qt4 porting (req. **FX4**).

One major short coming of the the DUChain is its API, which is highly focused on the editor use case. This use case can be simply described as, given a location in a document return the declaration. Practical implementations of this can be found in code navigation and context information when hovering over certain code parts. Also, only returning the location of an use is not very helpful when different uses of the same constructs can have different properties which may influence the transformation.

1 More detailed design information can be found at http://api.kde.org/extragear-api/sdk-apidocs/kdevplatform/language/duchain/html/duchain-design.html

In this chapter we gave a brief overview of the overal architecture of our automated porting system. Because the porting system is based on the KDevelop IDE, we described the components of KDevelop which are of importance for our work. We described the strong points of the IDE as well as the weak points that require extension to fit the needs of the porting system.

5

QUERYING

Source code contains many relationships between various elements of the source code which cannot be found without tools that understand these relationships. When looking for uses of functions that part of the API that needs to be replaced, one could use tools like Perl, AWK and grep. Regular expressions would be used to specify the name of the function and typical usage patterns, but this approach would fail to ensure in the first place that all uses are found and in the second place that the found uses are really uses of the function of interest and not one that happened to be defined in another scope.

Another issue in the context of transformations is that different uses of the same function might need different ways of porting. This happens when functions have default values for their arguments, resulting in uses of a function with a differing number of arguments between the uses. It also happens when the transformation depends on the value of an argument passed to a function call.

In this chapter we will describe the method used by the query engine which is build on top of the KDevelop framework. This query engine overcomes the deficiencies described in Section 4.2. We will start with general requirements specific for the query engine and than go into more detail on the various query types and the data structures representing the results of a query.

Querying is an essential component of a porting system, as it enables users to specify what they want to transform. Refining the requirements for the entire porting solution outlined in Chapter 1 to Chapter 3, we distill the following requirements for our transformation system:

QR1. *Ease of use* : the query system should allow users to easily specify which part of an API, used in their code, they actually want to transform. Such a specification should take a minimal effort.

QR2. *Genericity* : the query system should allow the user to specify a wide range of programming constructs as targets for their transformations. For example, users should be able to select only certain classes or methods of classes that comply with a specified signature.

QR3. *Customizability* : the query system should allow the users to specify constraints on the constructs they want to select for transformation. For example, a certain construct like the

call of a given method should only be selected if the value of one of the parameters adheres to a given constraint.

5.1 METHOD

We need a query mechanism which is adapted to the specific task of porting an API. Therefore it is designed in such a way that queries can be expressed in terms close to how an engineer looks at software in general and APIs in particular. An important issue influencing the design was that the query engine should have as little usage overhead as possible. Powerful tools are often laid aside by developers because the overhead of using them is just too high for the particular task at hand. As a well-known example, print statements are still often used for debugging while powerful, yet harder to learn, debuggers already exist for some time. What can be expressed in the language will therefore be limited, though enough to find the information needed for the task at hand.

A query can be formulated as a function Q accepting a `TopDUContext T` and an API specification `S`, returning a list containing hits `H` where the specified API is used:

$$Q(T, S) \rightarrow \{H*\} \tag{5.1}$$

The query system works by retrieving the `TopDUContext` from a given source file. It uses the DUChain API to find the construct (e.g. type or function) that is specified in the specification. Next it iterates over all the uses and filters out those which are not in the physical source file that is queried. The filtered uses are than used to find the corresponding places in the AST. The query engine than extracts further information needed for the transformation.

5.2 QUERY ENGINE DESIGN

The aim of our query engine is to find usages of specified elements of an API. This means that we do not want to construct an engine that can find any possible code pattern at any level of detail. Such engines exist, see e.g. the generic query engine implemented by Boerboom and Janssen [20] or SolidFX Telea and Byelas [35]. However, designing and implementing such an engine is highly complex and costly as outlined in by the previous two references. In our case we can reach our goal with less effort. Hence, we designed a lightweight wrapper around the KDevelop components which operates in terms related to API. I.e. the user can specify queries to find class uses, global function uses and method uses. This is reflected in the class diagram as shown in Figure 4.

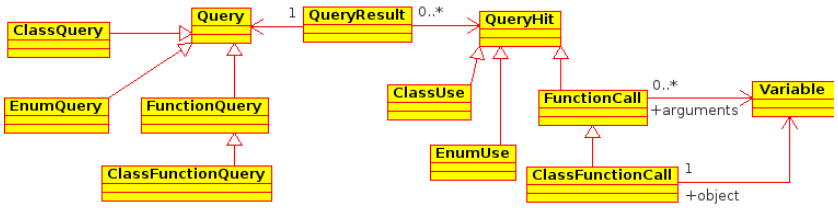


Figure 4.: Query and QueryHits class hierarchy

One or more queries can be performed on a source file resulting in zero or more hits. The query engine internally uses the translation unit, but only hits occurring in the queried source file are returned.

A hit is a data object which contains the actual location of one code pattern matching the query as well as various properties for that code pattern. Moreover, a hit provides access to particular parts (items) of the code pattern which are specific to the type of code element being queried. For example, a FunctionCall hit contains the items describing the locations of the hit's function name and parameters. Different types of hits have different types of items, loosely matching the structure of the AST.

The location or range of a hit is delimited by the first, respectively last, token of the code selected by a QueryHit in a source file. Ranges of the items of a hit are always subranges of the hit's range. The hit range and item ranges serve two goals:

1. They enable partial replacement of a hit. This ensures minimal impact and reduces the change of interfering transformations.

2. They enable partial reuse of the original code pattern, ensuring customizability of transformations.

In addition to items each hit type extracts specific information, i.e. the properties of a hit, from the AST that is relevant for the type. For example, for function and method calls the number of passed arguments is stored and for calls to constructors the hit stores whether or not it is an implicit constructor call. These properties form the base for conditional transformations, i.e. do different transformations for hits of the same type but with different property values.

5.3 QUERY TYPES AND RESULT CLASSES

Now we have outlined our general approach in the query engine we will go into more detail in the next sections on the currently supported queries. The `Query` class in itself is not usable, it only contains the unique identifier of a query which is used for presentation purposes mainly. The real queries are implemented as

subclasses of `Query`. We will outline the specific properties of and the results that are returned by each query type. In addition we will elaborate on the actual implementation details of the query engine here and special cases we encountered due to specifics of the DUChain.

The `EnumQuery` and its counter part `EnumUse`, depicted in Figure 4 will not be discussed separately. Both are straightforward, the query finds uses of the specified enum type or enum value and the hit only provides access to the range and does not have additional items or properties.

5.3.1 *Function queries*

A `FunctionQuery` lets the user search for calls to free functions, that is functions which are not class methods. A `FunctionQuery` is constructed with a Qualified Identifier (Qid), identifying the name of the function. In addition argument types must be specified for functions with one or more arguments. The return type of a function does not have to be specified as it is not part of the function signature according the C++ language definition. The query engine first looks up all function declaration for a given Qid. This returns either zero or more declarations, more than one in case of function overloading. Next the query engine compares the argument types of the query with the arguments of the found declarations. When a declaration is found, the query engine filters out all uses of the function that occur in the queried document. For each use an instance of `FunctionCall` is returned.

FREE TEMPLATE FUNCTIONS In addition to normal free functions we also added support for free template functions. To search for template functions, so called template restrictions must be set on the query. This can be an empty restriction, meaning that all instantiations for the template function with as many template arguments as restrictions are used to find uses. Template restrictions can also be non-empty which results in reported uses only for instantiations that have types as template arguments that match with the restriction set for the given template argument.

ARGUMENT-BASED RESTRICTIONS An important feature of the query engine in the context of porting is the ability to restrict queries for free functions and methods on the values of the arguments passed to function calls. For example, find only calls of function X where the first boolean argument has the value true. For this purpose we added two kinds of possible restrictions to the `FunctionQuery` class.

1. *Literal restrictions:* When a restriction of this kind is set on a `FunctionQuery` the literal value of the argument is matched against a regular expression. This works for all types of literals, e.g. string, boolean, integer and floating point, though all will be converted to a string internally and matched against the regular expression set as restriction. For numeric literals one can define a restriction like `1.23`, which will return calls with exactly this value. However, arithmetic restrictions, such as restricting numbers greater than some specified value, are not supported.

2. *Qid restrictions:* These were in the first placed introduced to restrict function and method calls on enum value usage. However, they can be used to restrict arguments on any name representing third-party or user-defined symbols.

Each argument in a `FunctionQuery` can have at most one restriction of each kind. When both a literal restriction and a Qid restriction are set for the same argument, the restrictions will be evaluated or-wise. One application for setting both restrictions can be found in boolean arguments. Often there are multiple ways to specify a boolean value, e.g. true, 1 and TRUE where the latter is some global type definition. To find all, or at least most, cases where true was passed one can set the literal restriction to `"1|true"` and the Qid restriction to `"TRUE"`.

There are some limitations to restrictions, however. First, no data flow analysis is done. Meaning, the restrictions are only applied to the actual text of the argument, where the text is either treated as a literal or as a symbol name. Second, it is currently not possible to restrict on macros. That is, restricting calls to functions or methods where a specified argument is some macro, is not possible.

FUNCTION CALL The result of a `FunctionQuery` is a `FunctionCall`. This class provides access to the ranges in the document of the various items of a hit. Items in case of `FunctionCalls` are the function identifier, template arguments if needed and function arguments. Ranges of individual items and the range of the complete use are stored in this class. It also provides access to the arguments of a function call by means of the `Variable` class. This class in turn provides some extra information to the user such as the type of the argument and the declaration of the type.

5.3.2 *Method queries*

Method queries are similar to the free function queries. However, as the name suggests, method queries only return calls to class methods. The reason for this distinction is that method calls are made on objects which requires different treatment when porting these kind of calls.

CONSTRUCTOR CALLS A special case in our context are constructor calls. A limitation of the DUChain which we did not solve yet is that it does not always report hits of `ClassFunctionQuery` in cases of constructor calls in contexts where conversions occur. When doing a look up of a function definition at a function call, KDevelop builds up the full chain of conversion steps for the arguments as needed. The problem however, is that this chain is not stored and therefore not accessible after the semantic analysis. This results in many unreported calls to constructors. This was left out for performance reasons and due to the limited usefulness in the main use case, i.e. editor support. This is an important drawback which should be kept in mind when using the framework for transformations while this problem is not solved.

CLASS FUNCTION CALL The `ClassFunctionCall` class provides some extra information in addition to the information provided by the `FunctionCall` class. In the first place it provides access to items that only make sense for method calls. For example, when a method is called explicitly on an object there are items for the object on which the method was called and the accessor, i.e. the token between the object and the call. This only returns a valid range when the call was explicitly on an object. Some more explicit examples of method calls and the items are shown in Listing 2. Second, in the case a method was called implicitly on `this` a flag is set to notify the user. Third, in case of constructor queries, it can tell if the constructor was called explicit or not.

Listing 2: Method call range and items

```
anObject->func( "someArg", 0 )
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ // Range
^^^^^^^^                       // Object item
        ^^                     // Accessor item
          ^^^^                 // FunctionId item
                ^^^^^^^^^      // Arg[0] item
                         ^     // Arg[1] item

anotherObject.func()
^^^^^^^^^^^^^^^^^^^^ // Range
```

```
^^^^^^^^^^^^        // Object item
             ^      // Accessor item
              ^^^^  // FunctionId item

// Call from method body of another method of the
// class. The range of accessor item is invalid.
// The property ImplicitOnThis is set to true.
func()
^^^^^^ // Range

SomeType::staticFunc()
^^^^^^^^^^^^^^^^^^^^^ // Range
^^^^^^^^              // Object item
        ^^            // Accessor item
          ^^^^^^^^^^  // FunctionId item
```

5.3.3 *Class queries*

A common use case when porting code between two versions of the same API is that classes get renamed. To support this case we added the `ClassQuery`. The `ClassQuery` only returns uses of classes and structs. To construct a `ClassQuery` it needs at least a Qid. The query engine first looks up the available declarations for the given Qid in the `TopDUContext` for the source file that is queried. In case of classes without a template this should be either one or no definition because of the so-called One Definition Rule (ODR) of C++ Stroustrup [33]. When a definition is found the query engine will request the uses for the definition and filters out all uses that are located in the queried source file. Finally, for each use found the engine creates an instance of `ClassUse`.

TEMPLATE CLASSES Template classes are quite common in many APIs. The `ClassQuery` therefore also has support for template classes. To search for template classes the same approach is taken as in the case of template functions. Template restrictions must be added to find uses of template classes with the same number of template arguments as restrictions set on the query.

CLASS USE The result of a `ClassQuery` is a so called `ClassUse`. A `ClassUse` is reported for each case where a type can be used. Usages of a class type are, for example, in base class declarations as types of function arguments as types of variable declarations or as part of qualifiers.

A special case is explicit constructor calls, i.e. constructor calls including the type name. These are considered as uses of a type to by the DUChain. Meaning that the snippet shown in Listing 3

will return three `ClassUse` instances when querying for class Test. It is important to keep this in mind when there are also queries defined for constructor calls. The type identifier item of both results will overlap, which is a potential cause for problems when defining transformations.

Listing 3: Type use due to constructor call

```
Test x = Test( ... ); // Two uses of Test
Test y( ... ); // One use of Test
```

In this chapter we discussed the design of the query engine which is part of the automated porting system. We first listed the requirements specific to the query engine, followed by a description of the method used in the query engine. Next, we described the various query types and query result types. An important feature of the engine presented in this chapter is the possibility to specify restrictions on arguments in function and method queries.

6 TRANSFORMING CODE

Minimal modification of the code is a highly desired property of the transformation engine. Each change made to code comes with the risk of breaking either the build or the run time behavior of the application. Moreover, too many changes make the code hard to understand by human readers, and recall, forty percent of the maintenance effort is spend on program understanding. In an ideal situation transformation tools would morph the code from one working state into the next working state. In practice however, this seems a goal which is hard to reach. Even small changes for which the transformations look straight forward can have surprising and unwanted side effects.

The aforementioned considerations lead to the following requirements for our transformation engine:

TR1. *Exact localization of affected places.* The engine should only change code that is contained in the results of the supplied queries Chapter 5, that is, not change any code which is not indicated as such.

TR2. *Minimal changes.* When keeping the changes as small as possible with a higher certainty of the correctness of the location, the actual consequences of the change will be better understandable.

In the remaining of this chapter we will briefly reiterate over the two major approaches for program transformation, discussed before in Chapter 3. Next we will present our source-to-source approach based on ranges of syntactical elements. Finally, we will give an overview of the design of the transformation engine.

6.1 METHOD

For the design of the transformation engine we have been inspired by the DMS system. In Baxter et al. [19] an approach is advocated that combines procedural transforms with with source-to-source rewrite rules. We will first discuss the approaches presented in Chapter 3 in relation to the stated requirements for the transformation engine. Next we will present our own approach.

6.1.1 *Procedural transforms*

The classical approach for transforming source is to do procedural transformations on ASTs which is also the case in Anisko et al. [17]. The first problem with Transformers is that the implosion of the parse trees into an AST strips out all layout information. Additionally all comments in the code are lost too because Transformers only works on preprocessed source. Another major problem of imploding the AST this way is that it brings the code in a state which is barely maintainable for human engineers due to the fact that it brings in the code of all included headers. Transformers therefore clearly violates our second requirement.

The approach of DMS to this problem is to adapt the preprocessor and parser in such a way that preprocessor symbols are passed to the parser first. The grammar of the parser is extended to expect preprocessor directives at statistically common places. Only when the directive could not be handled it is passed to the preprocessor and gets fully expanded. Procedural transformations can now be done while keeping the source in a clean state. It is noted in Baxter et al. [19] however, that this approach has its limitations due to the explosion of possible semantic meanings as result of macro expansion. A clean up step for macro uses before starting actual transformations, eventually making use of automated tools too is suggested.

6.1.2 *Source-to-source transforms*

Providing source-to-source rewrite rules in addition to procedural transforms has the advantage that the engineer does not have to get acquainted with the language at the level of a tree representation. This has a high practical engineering value because a lot of cases in Qt3 to Qt4 porting can be described in a relatively easy way when using source-to-source rewrite rules. This lowers the barrier for using the tool. We therefore decided to use source-to-source rewrite rules as the basis for our transformation engine. We did not provide procedural transform. This would require a similar approach as in DMS. However, this is not documented in a way such that it would be reproducible within the scope of this thesis work.

6.1.3 *Range based approach*

We need a transformation engine that only transforms source code at queried places and in addition does minimal changes to the source code. Therefore we designed a transformation engine that operates on query hits and only changes the (sub)range of a QueryHit. Our transformation engine directly works on the

source code of a program. In general transformations can be described as follows:

$$TX : S \rightarrow S' \tag{6.1}$$

That is, a transformation operates on a source representation `S` and returns a modified representation `S'`. To meet the first requirement stated in the intro of this chapter we use the results of the query engine described in Chapter 5. Furthermore, a transformation on source `S` for hit `H` results in a modified source `S'` where the modified range `r` is a subrange of or equal to the range of hit `H`.

We therefore define transformation TX as follows:

$$TX(S, H) \rightarrow \{S' = \{r\} | range(r) \in range(H)\} \tag{6.2}$$

where `range()` denotes the lexical range of a code element.

The transform engine works by creating an internal representation of the source file. For each hit in the file for which also a transformation is defined it tries to apply the transformation, making use of the information about the hit. It also does internal bookkeeping to cope with changed offsets due to performed transformations for hits later in the document.

6.2 TRANSFORM ENGINE DESIGN

As defined in Equation 6.2 a transformation uses a hit as guide for the range that should be transformed. The hits come from a query, which lead to the restriction that a transformation is always bound to one specific query.

A transformation is build up from one or more rules. A rule is either unconditional, the default, meaning that it is performed for each hit, or conditional, meaning that it is only performed if a given hit adheres to the condition of the rule. Each rule performs one or more specified actions. Currently we added support for insert actions and replacement actions.

Insert actions insert text, specified in the transform before or after the range that the transform is operating on. The kind of ranges a transformation can operate on is partly depending on the query kind. Replace actions replace the (sub)range of a hit on which the transform is operating with a text set in the transform. The text specified in the transform can also contain placeholders which refer to parts of the original text of the hit. This way transformations can be specified which replace (parts of a) hit while reusing some the original texts (e.g. a particular argument of a function call). The relations between the above described concepts are depicted in Figure 5.

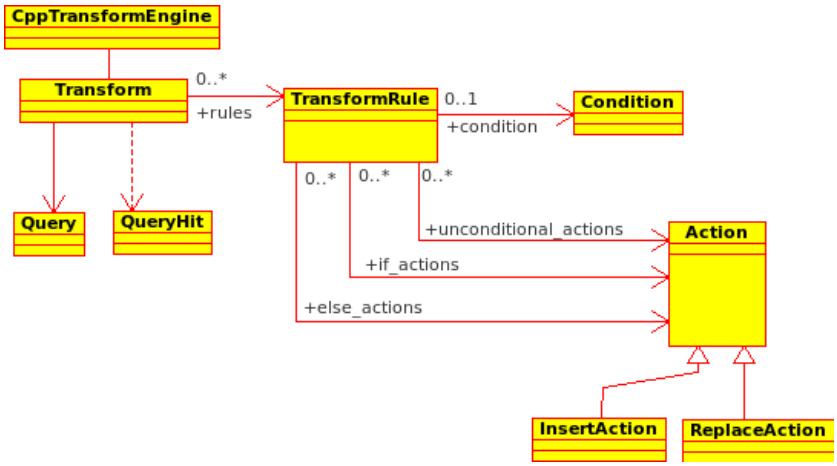


Figure 5.: Transform engine class hierarchy

This design gives quite some flexibility when specifying transformations. Especially when a hit has multiple items on which the transformation can operate. In those cases a transformation can be defined which either replaces the whole range or only specified items. Parts of the original text can be reused by means of placeholders. Another possibility is to define actions only for each item that needs to be replaced. This flexibility helps the transition of existing porting rules as it is more likely that the rules for our system can be defined in similar terms as the ones used before. To meet our customizability requirement we provided an `XML` format which can be used to specify transformations.

Although our transform engine purely operates on source files, we did not completely discard the option of procedural transformations (Chapter 3). The actions currently implemented directly change the actual text of the source, but actions that would do transformations in a procedural way could be added in future versions of the transformation engine. However, such transformations are more likely to break requirement **TR2**.

In this chapter we discussed the procedural and source-to-source approaches to program transformation. Next we discussed our range based approach to program transformation which was designed to meet the minimal impact requirement. Finally, we gave an overview of the design of our transformation engine that implements the range based approach for program transformation. In our design we have two main actions, insertion and replacement. Future versions can implement more complex actions to extend the possibilities of the engine.

7 USE CASES AND LIMITATIONS

Now that we have discussed the method and design of our query and transformation engines (Chapter 5 and Chapter 6), we will illustrate the usage of both with Qt3 to Qt4 use cases. This particular porting process was the original motivation for development of the porting framework. For each of the query types we will present one or more use cases, discuss the problems for these use cases with the current methods and discuss the limitations of our design where appropriate. In this chapter we use a simplified representation for clarity. The full XML format can be found in Appendix A and Appendix B. The presented examples are not meant to give an extensive overview of the rules needed for porting a Qt3 code base. They illustrate the most common rules needed for such a port though. For a more complete overview of the rules needed for a complete port of a Qt3 code base see http://doc.qt.nokia.com/4.6/porting4.html. We will conclude the chapter with a discussion on the limitations of our porting framework.

7.1 ENUMERATIONS

Even for enumeration renames it is necessary to have semantic understanding of the code. This is easily illustrated by a simple example. The official Qt3 to Qt4 porting tool does naive renaming of tokens in some cases when it is not run in strict mode. This results in transformations like:

```
mycolor.red()        -> mycolor.Qt::red(),
enum { Top, Bottom } -> enum { Qt::DockTop, Qt::DockBottom }
```

Just because these code constructs happen to have the same name in some local scope as Qt constructs that where renamed, it did not mean that they had to be transformed too. Alternatively, the tool can be ran in strict mode, resulting in less code constructs being ported, however, including the ones that should have been ported e.g. like the Qt::Red enum value. With this tool a trade-off must be made between the number of missed transformations in strict mode and the possible number of faulty transformations otherwise. In practice it is easier to fix the missed transformations by hand using the compiler errors than letting the tool do the job plain wrong at many places.

The faulty results of the porting tool, obviously is an unwanted side effect. Moreover, the engineer cannot know on forehand

what the precise side effects of the transformation will be and where these will occur. This means that additional time must be spend either on porting these specific constructs manually or on fixing the build afterward, without being able to know how much work that will be.

This problem becomes even worse when due to this approach the code still compiles but has undefined run time behavior for certain code paths. Unwanted implicit conversions are an often seen cause for such errors. One of the approaches to cope with these specific problems is commenting out the definition in the header of the source API. This clearly points out the problematic locations because the code will no longer compile. However, this directly shows a drawback of this approach, the code must be in a compilable state. Otherwise other compile errors may precede the ones that the engineer is actually looking for. Secondly, it requires the engineer to change all these places by hand which is definitely undesired for often used constructs in large projects.

For this reason we added support to find enum declaration and enum value declaration uses. Basically, both constructs can be found by their Qid. Queries to find enum uses can be specified as follows. This listing and the following show a simplified variant of the XML-based syntax for our query system. The complete syntax definition and concrete examples can be found in Appendix A and Appendix B.

Listing 4: Enum queries

```
// Query for enum declaration
enum-query
  uid="QButton::ToggleState"
  qid="QButton::ToggleState"

// Queries for enum values
enum-query
  uid="QButton::On"
  qid="QButton::On"

enum-query
  uid="QButton::Off"
  qid="QButton::Off"
```

Porting enum values in Qt3 is rather straight forward. In general either the class name, the enum name or a value has changed. The EnumUse class does not provide additional items. Meaning, an enum transformation always replaces the full range of the hit. Transformations for enumerators are like the queries fairly simple as shown in Listing 5.

Listing 5: Enum transforms

```
transform
  queryId="QButton::ToggleState"
  rule // Unconditional
   replace-action item="All"
   replacement-text="QCheckBox::ToggleState"

transform
  queryId="QButton::On"
  rule // Unconditional
   replace-action item="All"
   replacement-text="QCheckBox::On"

transform
  queryId="QButton::Off"
  rule // Unconditional
   replace-action item="All"
   replacement-text="QCheckBox::Off"
```

7.2 CLASSES

Qt3 contained various classes that where either renamed or replaced by other classes in Qt4. The class query is meant to deal with these changes. It does so in a generic way. The class query finds all uses of a class as a type, including inheritance specifications, class member specification, function argument specification and local variable declarations. Examples of these are the Qt3 classes `QIconSet`, `QWMatrix`, `QGuardedPtr<T>` and `QPtrList<T>`. Queries and transformations for these cases can be defined as shown in Listing 6.

Listing 6: Class renaming

```
class-query
  uid="QIconSet"
  qid="QIconSet"

class-query
  uid="QWMatrix"
  qid="QWMatrix"

class-query
  uid="QGuardedPtr<T>"
  qid="QGuardedPtr"
  template-argument

class-query
  uid="QPtrList<T>"
  qid="QPtrList"
  template-argument

transform
```

```
  queryId="QIconSet"
  rule // Unconditional
   replace-action
     item="All"
     replacement-text="QIcon"

transform
  queryId="QWMatrix"
  rule // Unconditional
   replace-action
     item="All"
     replacement-text="QMatrix"

transform
  queryId="QGuardedPtr<T>"
  rule // Unconditional
   // Only replace the type id not the
   // template spec of the hit
   replace-action item="TypeId"
   replacement-text="QPointer"

transform
  queryId="QPtrList<T>"
  rule
    replace-action
      item="All"
      replacement-text="QList<${TypeTemplateArg[0]} *>
```

Especially the two last transformations are interesting. They show to important functions of the transformation engine. The QGuardedPtr case demonstrates how to replace only part of a hit, i.e. the type identifier in this case. The QPtrList case demonstrates how to replace the complete range of the hit but reuse parts of the original code, e.g. a template argument, in the new code.

7.3 GLOBAL FUNCTIONS

The global function queries and transformations deal with all free, i.e. non-class functions. These functions can be either in the global name space or in another name space and can be both template functions and plain functions. Qt3 contains a global template function named `qt_cast`. Its precise signature is shown in Listing 7. From the documentation:

"...use the `qt_cast()` function to determine whether instances of QObject subclasses could be safely cast to derived types of those subclasses..."

Listing 7: Qt3 qt_cast signature

```
T *qt_cast<T *>(QObject *)
```

In Qt4 this function was renamed to `qobject_cast`. There are no special cases when porting this function so no restrictions are needed. The query for this function is shown in Listing 8. The `qt_cast` method is defined in the global name space so the qid of the method is simply equal to the function name. Argument types must be valid type expressions. These expressions will be validated in the context of the document that is queried to retrieve the node representing the type from the DUChain. Finally, we add an empty `template-argument` tag, as we want to find all uses of this function.

Listing 8: qt_cast query

```
global-function-query
  uid="qt_cast<T>(QObject *)"
  qid="qt_cast"
  template-argument
  argument
    type="QObject *"
```

Transforming `qt_cast` means that we only have to replace the function identifier of an use. In general, the FunctionCall class provide access to the ranges of the following items:

- FunctionId - The range of the function name in the hit.
- TemplateArg[i] - The range of the ith template argument. (Only for template functions.)
- Arg[j] - The range of the jth function argument.

We define the transformation for `qt_cast` query hits as shown in Listing 9.

Listing 9: Global function transformation

```
transform
  queryId="qt_cast<T>(QObject *)"
  rule // Unconditional
   replace-action item="FunctionId"
   replacement-text="qobject_cast"
```

7.4 METHODS

A large portion of the Qt3 to Qt4 porting work consists of porting changed function signatures. Mainly in this area we had to deal with more complex features such as restrictions on function arguments and conditional transforms. We will therefore elaborate a bit more on this type of queries. First we will present some examples related to porting of the `QString` class. These include

constructor call and operator porting. Next we will present some examples from `QObject` porting. Some `QObject` methods require different porting based on the arguments passed to the call.

7.4.1 *QString*

When porting a Qt3 code base to Qt4 there are two main problems related to the `QString` class. The first being that in Qt3 there where multiple ways to construct a QString from a `std::string`. There was a QString constructor and an overloaded `operator=`. Meaning that `QString` objects could be implicitly created from or implicitly assigned to from `std::strings`. In Qt4 this is no longer possible, `QString` objects can only be created from `std::string` by explicitly calling the static `QString::fromStdString()` method. The second problem is related to place markers in a char literal. Consider the following code snippet:

Listing 10: QString place markers

```
QString str( "Hello, %1, it's %1 today" )
  .arg( name ).arg( day );
```

In Qt3 this would have been valid code, in Qt4 however it is required that the place markers in the literal are consecutive numbers. The problem with this piece of code is that it does not result in a compile time error, so the run time behavior is silently changed without the engineer noticing it. Depending on the defined preprocessor symbols the latter can occur for the `QString( const char * )` as well as for the `QString( const std::string & )` constructor. Queries for both constructors can be defined as shown in Listing 11.

Listing 11: QString constructor queries

```
class-function-query
  uid="QString(const char*)"
  qid="QString::QString"
  argument
    type="const char*"

class-function-query
  uid="QString(const std::string &)"
  qid="QString::QString"
  argument
    type="const std::string &"
```

We did not implement support for replacing the faulty placeholders in a `QString` constructor call. However, the process of finding such calls can be speed up by using restrictions to make the query engine only find calls that have this faulty behavior. The query engine has support for literal restrictions. When a

literal restriction is set for a specific argument of a function, the query will only return a hit for a function call when its argument matches the restrictions.

Listing 12: Restricted QString constructor query

```
class-function-query
  uid="QString(const char*)"
  qid="QString::QString"
  argument
    type="const char *"
  restriction
    kind="LiteralRestriction"
    value="%(\\S+).*\\1"
```

Listing 12 shows the restricted query for `QString(const char *)` constructor. The query engine iterates all hits for the constructor and checks if the hit has a literal as argument. If so, the engine checks whether or not the lexical value of the argument matches the regular expression set in the restriction. Only those hits are returned where the lexical value matches the regular expression. This concept works for all literals (i.e. not only character literals but also boolean values and numbers).

To port the `QString` cases related to `std::string` we also need to find the overloaded `operator=( const std::string& )` of the `QString` class. Finding calls to overloaded operators classes works the same as finding normal functions as shown in Listing 13.

Listing 13: QString::operator query

```
class-function-query
  uid="QString::operator=(const std::string&)"
  qid="QString::operator="
  argument
    type="const std::string&"
```

Now the queries for the `std::string` cases are in place we define the transformations. For transforming the constructor hits we need two cases because a constructor can be called implicit or explicit as shown in Listing 14.

Listing 14: QString creation from std::string in Qt3

```
std::string stdStr;
...
QString qs1 = QString( stdStr ); // Explicit call
QString qs2( stdStr );           // Implicit call
QString s2 = stdStr;             // operator=
```

We can now define the transformation for all cases as show in Listing 15. The transformation is conditional on the ImplicitCtor call property which is one of the properties provided by the

ClassFunctionCall class. In the implicit case text is inserted after the `ObjectId` item, e.g. `qs2` in the previous code snippet. In the explicit case text is inserted after the `MemberId`, e.g. the second `QString` occurrence in the previous snippet.

Listing 15: QString std::string transformations

```
transform
  queryId="QString(const std::string&amp;)">
  rule
    if
      condition
        property="ImplicitCtorCall"
        expected-value="true"
      insert-action
        item="ObjectId"
        location="After"
        text="= QString::fromStdString"
    else
      insert-action
        item="MemberId"
        location="After"
        text="::fromStdString"

transform
  queryId="QString::operator=(const std::string&)">
  rule
    replace-action item="Arg[0]"
    text=QString::fromStdString(${Arg[0]})
```

7.4.2 *QPtrList*

We already introduced the template class called `QPtrList`, which stores a list of pointers to objects of the type passed as template parameters. In Qt4 this class is replaced by the more generic `QList` class. Although the API is somewhat similar there are some differences which needs porting. In order to find functions of template classes we needed a way to specify the type of a functions argument which is the same as the template type specified for the class. This is done by means of a place holder as shown in Listing 16. In addition it is possible to specify a real type which limits the result to hits of instantiations for that specific type.

Listing 16: QPtrList<T>::containsRef

```
class-function-query
  uid="QPtrList<T>::containsRef"
  qid="QPtrList::containsRef"
  const="true"
  template-argument
  argument type="const ${templateArg[0]} *"
```

```
transform
  queryId="QPtrList<T>::containsRef"
  rule
    replace-action
      item="FunctionId"
      replacement-text="count"
```

7.4.3 *QObject*

The QObject class is an important part of Qt because it is essentially the basis for the object framework of Qt. QObjects organize themselves in trees and provide features such as object communication through the signal/slot mechanism, events and event filtering and automatic deletion of child objects. One of the reasons for changed run time behavior after a Qt3 to Qt4 ports is related to string comparisons. An example of this is that in Qt3 meta information could be used on QObject based objects to check for inheritance information:

Listing 17: String based comparison in Qt3

```
// bool QObject::inherits(const char *clname) const
QTimer *t = new QTimer;   // QTimer inherits QObject
t->inherits( "QTimer" );  // returns true
t->inherits( "QObject" ); // returns true
t->inherits( "QScrollView" ); // returns false
```

A particular problem that raises due to string comparisons is that name changes do not result in compile errors but do change the run time behavior. In Qt4 for example, `QScrollView` was renamed to `Q3ScrollView`. So code relying on calls like `QObject::inherits( "QScrollView" )` returning true, will now silently fail. When transforming these kind of API calls the engineer will clearly take the string passed to call in account.

Listing 18: QString std::string transformations

```
class-function-query
  uid="QObject::inherits(QScrollView)"
  qid="QObject::inherits"
  const="true"
  argument type="const char *"
    restriction
      kind="LiteralRestriction"
      value="QScrollView"

transform
  queryId="QObject::inherits(QScrollView)">
  rule>
    replace-action
```

```
      item="Arg[0]"
      replacement-text="\"Q3ScrollView\""
```

The final use case we present here comes from porting `QObject::child` for which the signature is shown in Listing 19. In the Qt3 this method searches for children of the object on which it is called which have the given object name. Optionally the search can be limited to objects that inherit from a given class. By default the search is recursive which means that a depth first search in the object tree is done. In Qt4 the child method does not exist any more and depending on the argument values the hits have to be ported in different ways.

Listing 19: QObject::child signature

```
QObject *QObject::child(const char *objName,
                        const char *inheritsClass = 0,
                        bool recursiveSearch = true);
```

The recursive search variant of calls to this function have either one or two arguments. It can be that the call has three arguments with the value of third argument set to true. In practice however, this variant of recursive search is hardly used. We therefore do not cover the case but similar queries could be defined to handle this case as well. In the cases where child is called with a non constant third argument, data flow analysis would be needed to actually discover whether the call is recursive or not. Data flow analysis is not covered by the requirements and therefore not implemented.

Listing 20: QObject::child recursive variant

```
class-function-query
  uid="QObject::child"
  qid="QObject::child">
  argument
    type="const char *"
  argument
    type="const char *"
  argument
    type="bool"

transform
  queryId="QObject::child"
  rule // foo->child("objName")
       // becomes
       // qFindChild<QObject *>(foo, "objName")
    if
      condition
        property="ArgCount"
        expected-value="1"
      replace-action
```

```
      item="All"
      replacement-text=
        "qFindChild<QObject *>"
        "(${ObjectId}, ${Arg[0]})"

rule // foo->child("objName", "Class")
     // becomes
     // qFindChild<Class *>(foo, "objName")
  if
    condition
      property="ArgCount"
      expected-value="2"
    replace-action
      item="All"
      replacement-text=
        "qFindChild<${literalVal(Arg[1])} *>"
        "(${ObjectId}, ${Arg[0]})"
```

There is no need for a special query for the recursive variant. We assume particular usage of the API and define separate rules for the one and two argument cases in the transformation. Because the calls of the non-recursive variant will always have three arguments, these will not be changed by the transformation specified in Listing 20 even though these cases are part of the query result.

For the recursive searches the complete hit must be replaced with a call to the global Qt4 template function `qFindChild`. When no second argument is given the template parameter should default to `QObject *` and in the case that the second argument is given the template parameter should be set to that value.

To support these transformations we added a property to `FunctionCall` called `ArgCount`. Additionally, support for functions in the placeholders was added. The function needed here was one to retrieve the unquoted argument value in cases where the second argument was set. Listing 21 shows the query and transformation for the non-recursive change. Two features of the transformation language shown here are accessing items that have an offset and the literalVal function. The latter will try to remove quotes from the beginning and the end of the range of the item passed to it or pass the lexical value unchanged otherwise.

For the non-recursive variant a query is used which has restrictions on the third argument as shown in Listing 21. These restrictions work or-wise, meaning that either calls with the qualified identifier `FALSE`, which is a type definition in one of the Qt3 headers, or calls with a literal `false` passed as third argument are found. The non-recursive variant is also special in the sense that there is no replacement in the Qt4 API for doing non-recursive searches in object trees. The general approach in porting

projects is that a custom global function is provided which has this particular behavior.

Listing 21: QObject::child non-recursive variant

```
class-function-query
  uid="QObject::child (non recursive)"
  qid="QObject::child">
  argument
    type="const char *"
  argument
    type="const char *"
  argument
    type="bool"
    restriction
      kind="QidRestriction"
      value="FALSE"
    restriction
      kind="LiteralRestriction"
      value="false|0"

transform
  queryId="QObject::child (non-recursive)"
  rule
    action
      type="replace"
      item="All"
      text="Util::findDirectChild"
           "<${literalVal(Arg[1])} *>"
           "(${ObjectId}, ${Arg[0]})"
```

7.5 LIMITATIONS

While building up the transformations for Qt3 to Qt4 porting we discovered some limitations that our porting framework has. Some of these limitations are structural and inherent to the approach we took. Others are minor limitions and would either require some more implementation effort to get it right or could be left as is as a trade off. We will elaborate on the structural limitations in this section and briefly mention some of the less severe limitations.

7.5.1 *Structural limitations*

Due to the fact that we take single API elements as base for our porting framework it can occur that two queries have hits with ranges that overlap. A trivial example is shown in Listing 22. When the defined transformations for both queries, both transform the complete range of the hits, clearly a conflict occurs. In this particular case, careful specification of the transformation

can avoid the conflict. In general however it cannot be guaranteed that such a workaround exists. In addition, even if such a workaround exists, it cannot be assumed that the user of the porting framework is aware of all possible conflicts that might occur for the queries and transformation he specifies.

Listing 22: Overlapping query ranges

```
QGuardedPointer<QScrollArea> qsap;
^^^^^^^^^^^^^^^^^^^^^^^^^^^^ // QGuardedPointer range
               ^^^^^^^^^^^^  // QScrollArea range
```

A limitation related to this can be experienced when having a chain reaction of changes. One example we found was the return type of a method. The particular type the method returned according to the declaration was `QObjectList*`. The body of the method returning this type had a return statement which returned the value of `QObject::queryList()` as shown in Listing 23.

Listing 23: Changed return value

```
class SomeClass : public QObject
{ ... };

QObjectList *SomeClass::someMethod()
{
  ...
  // QObject::queryList
  return queryList(...);
}
```

With our approach a hit would be found for `QObjectList` in the line four. Note, the pointer specifier `*` is not part of the range in our query engine. This means that when the transformation for `QObjectList` is performed, the return value is a `QList<QObject *> *`. Next, the engine will find a hit for `QObject::queryList` in line eight. This call will be replaced with a call to the free function `qFindChildren`. The resulting code would look something like Listing 24.

Listing 24: Changed return value ported

```
class SomeClass : public QObject
{ ... };

QList<QObject *> *SomeClass::someMethod()
{
  ...
  // QObject::queryList
  return qFindChildren<...>(...);
}
```

The problem now is that the `qFindChildren` does not return a pointer to a `QList<QObject *>` but returns it by value. There is currently no way to express these kind of chain reactions in our framework and will need manual correction afterward.

In general we can say that changed return types are not handled very well by the porting system. If we take the previous `QObjectList` example one step further we encounter another problem. Lets assume that the system is able to handle this chain reaction, thus it can change the return type to be by value. The next problem is that for every location where the return type is assigned to a variable, the engine would need to support change of access to the variable too. The best case is when the return type only changed from return by pointer to return by value. Worst case scenario is a complete different return type. In the latter case not only the types of the variables, to which the result of the calls to this particular function are assigned, must be changed but also all calls on these variables might have become invalid at this point.

Finally, we discovered that our porting framework works really well for local changes but cannot handle more structural changes of code. Meaning, changes that require more than just replacing or modifications of a hit cannot be performed in a clean and generic way. We encountered cases in Qt3 to Qt4 porting that require for some parts more structural changes. We will discuss the structural changes required for `QMainWindow` constructor calls.

In Qt3 the `QMainWindow` constructor had a bit flag argument to control various properties of the window. These bits can be set using enum values. A typical example of this is shown in Listing 25.

Listing 25: QMainWindow flags in Qt3

```
class MyWindow : public QMainWindow {
  MyWindow()
    : QMainWindow(..., Qt::WDestructiveClose)
  { }
};
...
QMainWindow *w = new QMainWindow(... ,
                        Qt::WDestructiveClose);
```

In Qt4 some of these values where split out or removed and setter methods on `QMainWindow` must be called to get similar behavior as shown in Listing 26. It is clear that not only the text ranges containing the constructor call are changed, but additional text is inserted at specific locations depending on the context.

Listing 26: QMainWindow ported to Qt4

```
class MyWindow : public QMainWindow {
```

```
  MyWindow() : QMainWindow(...)
  {
    setAttribute(Qt::WA_DeleteOnClose);
  }
};
...
QMainWindow *w = new QMainWindow(...);
w->setAttribute(Qt::WA_DeleteOnClose);
```

7.5.2 *Minor limitations*

Besides the more structural limitations as discussed in the previous section, there are also some less severe limitations which we will briefly mention here. The first thing is that XML is not a very convenient way to express queries and moreover transformations. However, we made a trade off here on purpose between ease of use and ease of processing. There exists various frameworks for parsing XML on top of which special purpose parsers can be build quite easily. Also, there exists various tools for validating XML files. To partially overcome this problem we added XML extraction support in the context menu. The developer can right click a class or method to extract the query for that particular API element.

Another limitation is that currently the porting framework has no support for preprocessor symbols. Obviously this is something that is of particular interest in a porting framework, e.g. changed header names or macro definitions. However, the fact extraction engine of KDevelop does provide access to this kind of information, we therefore think that adding support for preprocessor queries and transformations is relative easy.

For performance reasons KDevelop does not re-parse all included headers but reuses information stored in the DUChain. This does not always work correctly as we found out in the case of `QString` and the `QT_NO_CAST_ASCII` preprocessor symbol. When this macro is defined during build, the `QString( const char* )` constructor will be disabled. This means that there will be an implicit conversion to `std::string` and thus calls to `QString( const std::string & )`. If the first document parsed contains this macro, the header for `QString` will be parsed and stored this way in the DUChain and might therefore contain incorrect information for other source files which do not have this macro defined. The workaround is to tell the parsing framework to always parse everything which comes with a performance penalty which is, at least in the case of Qt3 to Qt4 porting, not always required.

In this chapter we demonstrated the usage of our automated porting framework by presenting various examples from the Qt3

to Qt4 porting use case. We showed queries and transformation descriptions for enumerations, classes, global functions and methods. The examples included demonstration of important functionality such as function queries with restrictions on arguments and different transformations for the same query based on the hit properties. Finally, we discussed the limitations of the porting system, the most important ones being the risk of interference between two or more transformations, the lack of being able to deal properly with changed return types of functions and the fact that the engine cannot do transformations that require changes to the context of a query hit very well.

8 VISUAL SUPPORT FOR ESTIMATION AND PORTING

An important aspect of software maintenance is program understanding. In our specific context the main task in the understanding process is twofold. At a coarse level the engineer wants to gain insight which parts of an API are used in a project. This information is important when making project estimations as well as for keeping track of the porting progress over time.

On the other hand at a fine grained level the task is to gain insight in where a particular file is affected by the specified transformations. This is in particular important due to the limitations of the transformation engine. Transformations might interfere with each other as mentioned in Section 7.5.1. Therefore the engineer needs a way to identify these conflicts before the transformations are actually performed. Besides gaining insight in how transformations interact with each other it is also important to get an idea how the actual code is affected by the transformations.

For these two tasks we need several levels of detail showing information about the project that is to be ported. The first task requires an aggregated view showing how the source API is used within a project. The second task needs a more detailed view on file level showing structure of the code. First of all the source code itself must be visible. Next to that an overview of the available transformations for a specific source file is needed. The last view needed is one showing where the transformations will modify the file and if there are transformations that interfere with each other. These views must support the engineer in a simple and intuitive way which integrates tightly with his workflow.

Hence, we propose to add software visualization techniques to our plugin for the KDevelop IDE. We need visualizations that are scalable for large amounts of data. Furthermore we are not so much interested in relational visualizations such as graphs because our focus not on relations between code constructs but between code and transformations. Therefore we reused and adapted dense pixel techniques.

As outlined in Chapter 1, the most important motivation for this project was Qt3 to Qt4 porting. Various use cases can be identified related to porting projects:

- Estimation of a porting process.
- Performing a port.
- API feedback and refactoring estimation.

- Deprecated API tracking.
- Identify which parts of a class are affected.
- Get insight in the complexity of code that will be affected by a transform.

In the remaining of this chapter we will use these use cases as a guide to present the visual features we added to the plugin in order to support the user while performing these tasks. In addition, videos demonstrating the features presented in this chapter can be found at:

- http://www.youtube.com/watch?v=_j0-PTe04ow
- http://www.youtube.com/watch?v=9CxXiQ30ghY

8.1 USE CASE: ESTIMATION OF A PORTING PROCESS

Before a Qt3 porting project starts an overview of Qt3 API usage in the code base that is subject to porting is needed to estimate the amount of work needed for the port. Especially important is to get an overview of how the work is distributed over the files. When many files are equally affected, the workload will be higher than when only some files are affected. Moreover, it will become harder to predict the side effects of a port.

To illustrate the usage of our plugin we used the porting file as found in Appendix B on the code base of the latest Qt3 based release of kdelibs package from the KDE Software Compilation (KDE SC)[1]. This package contains about 750 KLOC, uses a wide range of the Qt3 API and is therefore an interesting show case for our framework. To this extend we

- describe the initial setup when using our tool (Section 8.1.1),
- introduce and describe the project overview (Section 8.1.2),
- desribe how to interpret the resultes presented by the project overview (Section 8.1.3),
- describe how the project over view can be configured to present data in different ways (Section 8.1.4).

1 The used version is 3.5.10 and can be downloaded from http://www.kde.org/info/3.5.10.php

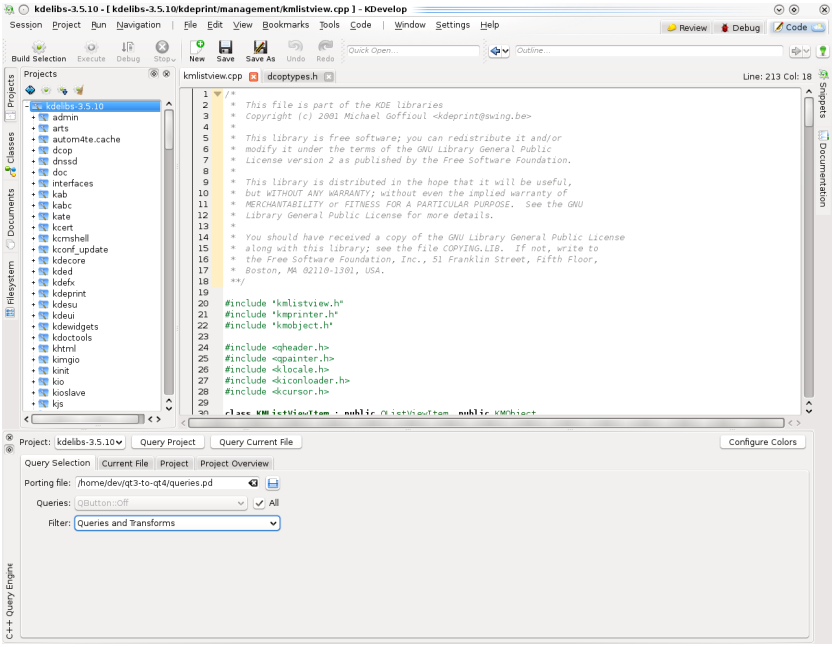


Figure 6.: Overview of KDevelop with our plugin enabled.

8.1.1 *Initial setup*

Initially the user opens the project he wants to work on, i.e. kdelibs in our case. KDevelop mostly resembles the typical IDE as can be seen in Figure 6. On the left the project tree is enabled, furthermore there are a class browser, open document browser and a file system browser. The center contains the source editor and in the bottom we placed our plugin. The first tab of the plugin, of which a detailed view is shown in Figure 7, offers the possibility to open a porting file containing the queries and transformations (Figure 7 - 1). Additionally, the user can configure the query selection once a porting file is opened.

This selection determines which queries are executed when a document or project is queried. First, he can choose to select either only queries without a transform, queries that have a transform or both (Figure 7 - 4). Being able to perform only queries supports the case of estimating the work for constructs that are known to have partial transformations, e.g. due to changed return types as described in Section 7.5.1, or no transformation at all. Second, he can choose if all queries (Figure 7 - 3) of the current selection or only the currently selected query (Figure 7 - 2) must be performed when a document or project is queried. When the selection is configured a query can be started for the currently selected project, or if a source file is open, a query for the current source file.

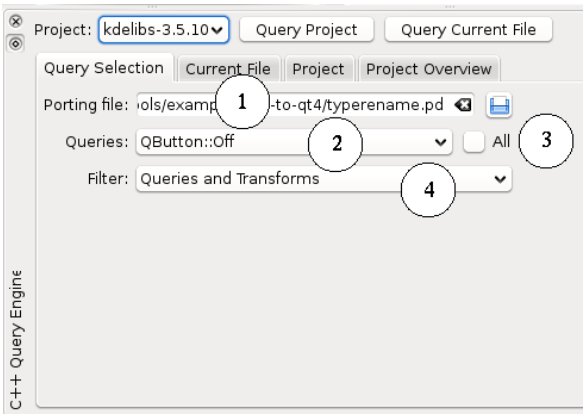


Figure 7.: Query selection tab.

8.1.2 *Project overview*

Finding uses of symbols using typical IDE built-in search functions (see Section 3.6.3) is a start for a solution for the estimation task. Assuming that the IDE implements this query using semantic information, more precisely scoping and correct symbol lookup, the results are more precise than when using grep like tools. However, for estimation purposes this would require a lot of manual work because the IDEs only support one query for uses at a time. For estimation purposes an aggregation of this data is needed. This aggregated data should in addition be presented to the user in such a way that it is easy to understand and helps estimating the porting effort.

For this use case we selected all queries, i.e. queries with and without transforms, and started the query for the whole project. While the query is being performed in the background, the user can switch to the project overview tab which is shown in Figure 8. For an even better overview, the plugin window is maximized. This view is updated every time a file is parsed and queried until all project files are processed.

The results shown in Figure 8 give an overview for all queries as described in the porting file in Appendix A performed on the kdelibs code base. There where in total about 2177 hits for 15 different queries in 569 different files. On the left side controls are available to manipulate which data is shown and how it is shown in a panel that can be hidden. Figure 9 shows the project overview with the control panel visible. On the right side the data is presented in a view based on the table lens as described in Pirolli and Rao [29] and Rao and Card [30] and on the extended table lens as presented in Telea [36].

By default the data is ordered and filtered to support porting estimation. The file names are set as row labels and the query identifiers as column names. Only files that are actually pro-

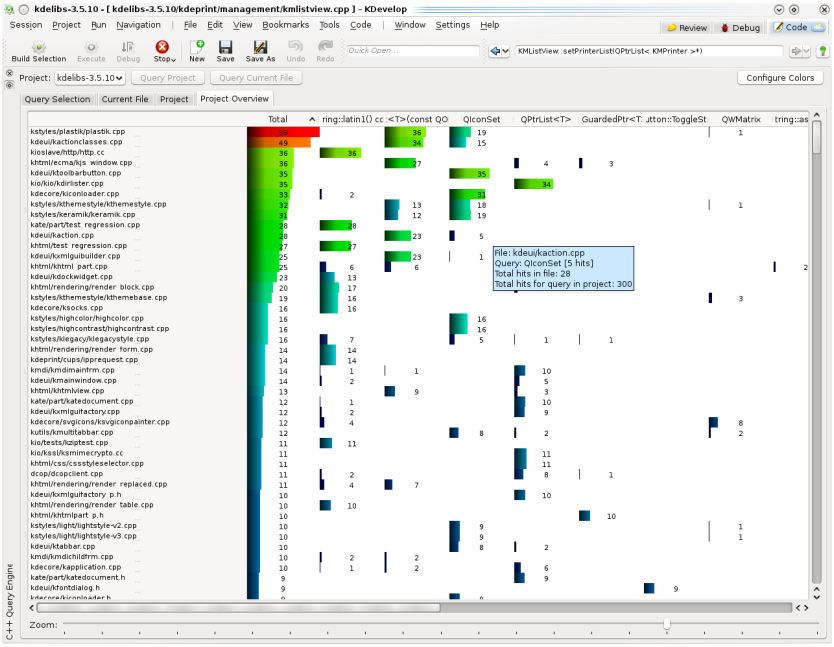


Figure 8.: Project overview for kdelibs.

cessed, i.e. C++ files and headers, are shown. Other files like documentation and sources in other languages are ignored by the plugin. The first column of the table contains the total number of hits for the files. All consecutive columns contain the number of hits for the query represented by the particular column.

The length of the bars for the hits of specific queries is proportional to the total number of hits for the file represented by the row. For the color of the bars we used a rainbow color map, implemented as a function which calculates the value when needed, applied to the normalized values of the column. A zero value maps to blue and one to red.

Empty rows, i.e. files for which none of the queries returned results, are filtered out. Empty columns, i.e. queries for which no hits where returned in any of the files, are filtered out too. The table can be sorted by column by clicking on the column header and is sorted on total by default. When the user hovers over the colored bars, a tool tip is shown with the specific data for that cell, e.g. file name, query id, hits for the query and total hits in the file.

8.1.3 *Interpreting the results*

Given that all queries relevant for porting are specified, which is not the case in our examples as we only specified a subset of the queries needed for a complete Qt3 to Qt4 port, this view can help answering the questions related to estimation as follows. First the table is sorted on the total column. The distribution of the

total column now tells something about how the porting work is distributed over the files of a project. In Figure 8 we see for example a positive skew. This is interpreted as that most of the porting work is located in approx. five percent of the files in the project. If the distribution were normal or even worse it would mean that for the porting, a larger percentage of the project files, would need about the same amount of changes and recall that a large number of changes makes the resulting code harder to understand.

Next we look at the distribution of the individual queries. The results are interpreted in the same way, however an extra dimension is added when taking in account that the developer has knowledge on what the queries represent. Some queries might represent API that only requires minimal effort, e.g. renaming of an enumeration value. Other queries however might indicate that concepts are used which are deprecated and must be ported to the new concept. Qt4 introduced a new model/view framework which requires a lot of effort when a Qt3 code base must be ported to the new concepts of that framework. The distributions of the individual queries can therefore be used to allocate the developers with the required knowledge for the specific task.

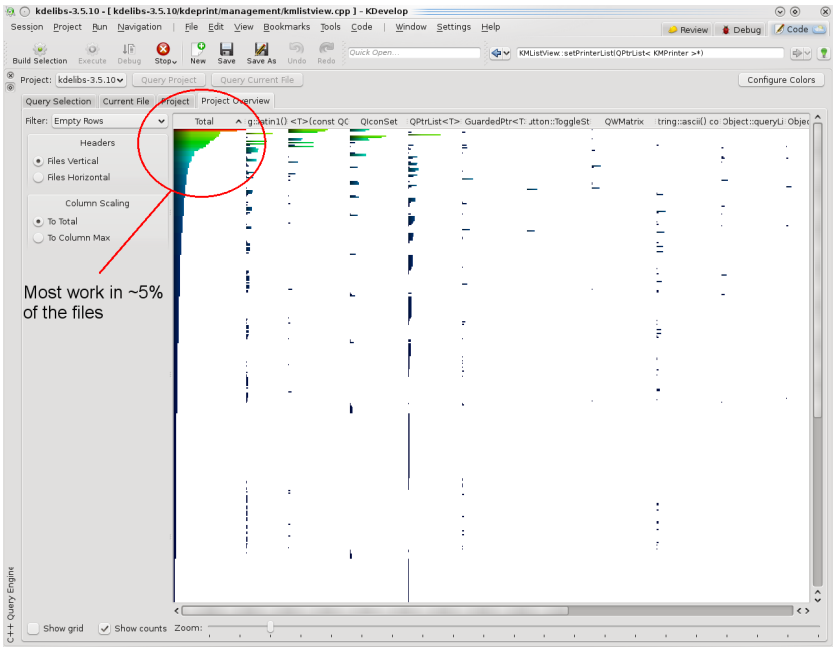


Figure 9.: Project overview for kdelibs zoomed out.

In a last step the distribution in the total column and the distribution of an individual query can be combined to estimate the amount of time a developer must be allocated with respect to the total time available for the project. In Figure 9 we see the same overview as in Figure 8, but zoomed out to get an overview for all files in the project. We see that most of the work in the

top five percent of the files is the sum of the first four queries. We know that the first three queries have transformations which can be performed without causing additional work. The fourth one, a query for `QPtrList`, however, is known to result in manual work afterward. It can be seen that this query effects quite some files in the project and will therefore have a higher impact on the needed time than the three queries before.

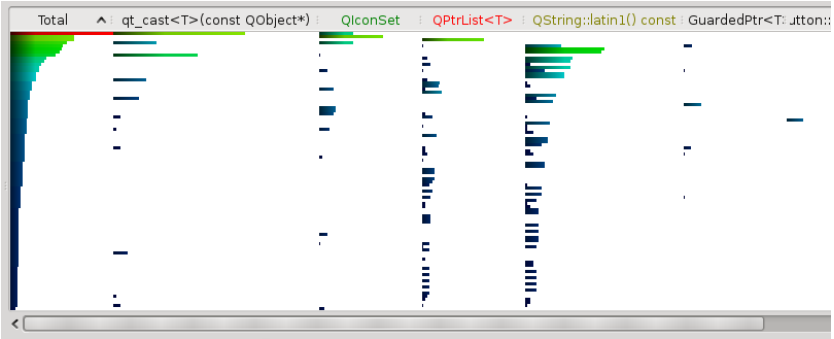


Figure 10.: Header coloring by scale.

To support the user even more with this kind of reasoning he can specify a scale to a query, where the scale is one of easy, medium, hard. When queries have a scale assigned the color of the header text in the table lens is set to green for easy, dark yellow for medium and red for hard. This way the user gets a visual hint on the difficulty of the queries and is therefore able to reason in a more insight full manner about the effort and risks of a port. This concept is demonstrated in Figure 10. When no scale is defined the headers text color defaults to the color defined in the system palette.

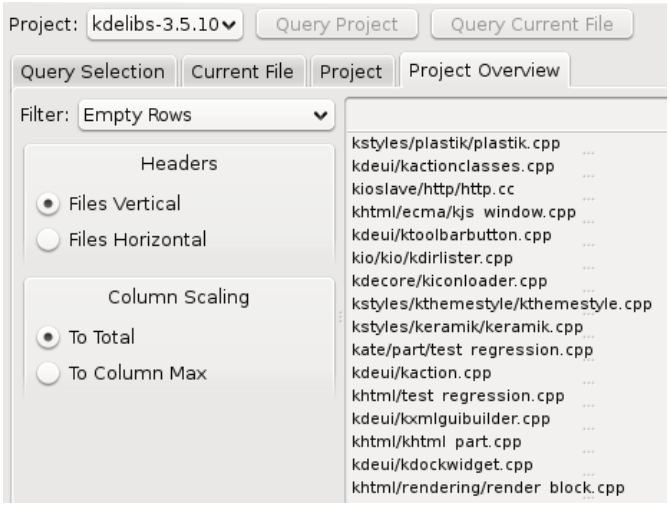


Figure 11.: Data presentation control panel.

8.1.4 *Configuration of data presentation*

Besides the default representation of results in the project overview tab there is a panel left of the table lens, shown in Figure 11, containing controls to adjust the way the data is presented. This panel can be hidden to maximize the area for the table lens, as is the case in Figure 8. The filter combo box enables the user to make empty rows and empty columns visible. By default, the bars in the columns for the queries are scaled to the total column. However, to get a better view of the distribution of specific queries the bars can also be scaled to the columns maximum. Some users might prefer to have a visible grid, therefore the grid can be enabled. When the row height is large enough, the counts are shown in the middle of the bar. These can be disabled in case only a global overview is needed. The zoom slider can be used to adjust the tables row heights. This is particular important for projects that have a large number of files. When zooming out visibility of the grid and show count will get disabled in the table view when the used font with a size of 6pt does not fit anymore. They will get enabled again when zooming in to a size that allows to fit the font again.

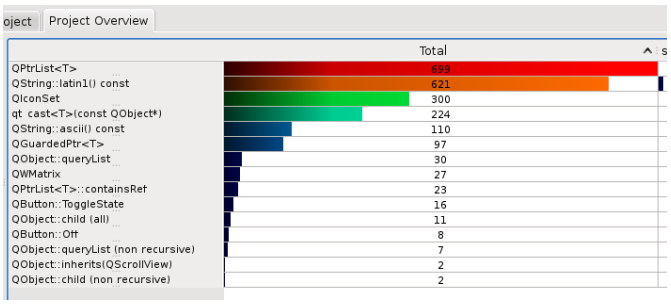


Figure 12.: Detail of the project overview with rows and columns swapped.

The last option is the possibility to swap rows and columns, of which a detail is shown in Figure 12. This option interchange the row headers and the column headers. This changes the meaning of the table as follows. Before the exchange each cell `(i,j)` represents the number of hits for query `j` in file `i`, except the cells in the first column which represent the total number of hits for file `i`. After the exchange each cell still represents the number of hits for a query in file, with `i` and `j` flipped. The total column however, now represents the total number of hits for query `i`.

The total column, now gives an indication of the distribution of the usage of the queried API in a project. This information can be used in several ways. First it gives insight in how the queried API is used in the project. For porting projects this gives a quick overview of the possible pitfalls during a port, i.e. it quickly

shows how much API that is hard to port is used in the project. Second, from an API designing perspective, this view can be used as a guide to reconsider the grouping of the API into libraries. I.e. queries for which a high count is reported might point to classes and functions that might have a logical relation. It might make sense to group them in a single library or component when this is not already the case. A third use case of this view is related to optimization. When having different projects available the use the queried API an aggregation of the total distribution for these projects can be used to prioritize the areas in the API for which optimization would have the highest impact. Having one slow function that is only used sparely in projects might not be worth the effort and risk of changing it. However, having a function that is used very often might be worth the effort of optimizing it.

8.2 USE CASE: PERFORMING A PORT

Besides supporting the task of porting effort estimation the plugin also aims to support the actual porting task. In general the task of porting consist of navigating through the projects source files, finding the locations that need porting and performing the actual change. In some cases it might be possible to do this automatically in other cases not. Some parts of the code can be so complex that it is preferred to do no automatic changes at all. A limitation of our transformation engine is that two or more different queries might have hits at ranges that interfere with each other as described in Section 7.5.1. In this section we present a new technique that helps the developer to quickly identify where queries have hits in a file. In addition the used visualization helps to identify possible interference between queries.

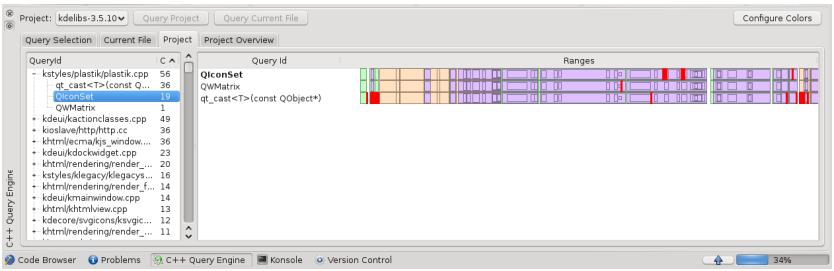


Figure 13.: File oriented result browsing.

8.2.1 *File oriented view*

Figure 13 shows the file oriented view of the plugin while a query is still running on the kdelibs code base. On the left side a tree view is shown which gives a coarse overview of the result.

For each file that has hits for one or more queries an item is shown in the tree. Next to the file name the total number of hits in the file is shown. The tree is sorted descending on count by default, resulting in the files which have most hits and thus most work appearing at the top of the list. For each query that has hits in a particular file a leaf is added to the item in the tree for that file. Next to the query identifier the number of hits for that particular query is shown. This organization makes it easy to get a quick overview on how the work is distributed within the file the developer is working on with respect to the queries that have hits in that file. The tree view is tightly integrated with the rest of the IDE. Single clicking items, either the file items as well as the query items, directly updates the widget right of the tree view. Double clicking items will also make the editor opening the file related to item that was clicked.

Right of the tree is a widget which we call the file impact view. It is a new technique to visualize where queries have hits in a particular file within limited space. The view shows the results for the file selected in the tree. Each colored bar represents the results of a different query for the same file. The places where queries have hits in the file are marked by a red patch. For example, Figure 13 shows that there are hits for the three queries `QIconSet`, `QWMatrix` and `qt_cast` in the file `kstyles/plastik/plastik.cpp`.

8.2.2 *File impact view rendering*

The colored bars shown for each query, depict a simplified representation of the source code in the queried file. The main idea for this representation is loosely based on the well-known SeeSoft tool Eick et al. [23] and the visual code navigator Lommerse et al. [27], both using a dense pixel technique for representing source code. The AST, representing the syntactic structure of the source file, as produced by KDevelop's C++ parser serves as model for the colored bars. We render the simplified representation of the syntactic structure of the code in a colored as follows. First a color mapping is loaded from a stored configuration, which contains a mapping between AST nodes and colors. Next, we visit the AST in depth first order using a visitor implemented for this particular purpose. Each AST node is rendered as a distinct colored block. AST nodes that appear on the same level in the tree result in blocks of the same height, ordered from left to right, as the nodes they represent come in the source file scanned from beginning to end. The width of each block is proportional to the number of characters of the syntactic construct it represents in the actual source. Nested code constructs get a height assigned which is a fixed amount smaller than the height of the parent, and are centered vertically in the parent. Space separating consecutive blocks

AST NODE	ENABLED	COLOR
ClassSpecifierAST	yes	Green
CastExpressionAST	no	-
CatchStatementAST	no	-
DoStatementAST	yes	Brown
ForStatementAST	yes	Brown
FunctionCallAST	no	-
FunctionDefinitionAST		
- public method	yes	light green
- protected method	yes	light orange
- private method	yes	light red
- free function	yes	cyan
IfStatementAST	yes	Purple
SwitchStatementAST	yes	Purple
SignalSlotExpressionAST	no	-
StringLiteralAST	no	-
TryBlockStatementAST	no	-
WhileStatementAST	yes	Brown

Table 1.: Default AST nodes configuration

at the same level of nesting is proportional with the amount of code present between the code in those blocks and located in constructs which are not selected for visualization. Each block is colored in a hue which indicates the type of its AST node taken from the color mapping. Blocks for AST nodes that have no color mapping and blocks that have a height or width smaller than a fixed limit are not selected for visualization. Finally, for each hit of the query a red patch is rendered as follows. The height of each patch is always the full height of the bar, regardless the level of nesting where the code construct of the hit occurs. The length of the patch is proportional to the number of characters represented by the code construct, unless it gets smaller than a fixed width to prevent the red patches to become invisible for large source files.

The color mapping is stored in an user specific configuration file, i.e. the same configuration will be used for all projects the user works on. When no configuration exists, e.g. on first use, a default configuration is created. Table 1 shows the default configuration for the supported AST nodes. The configuration can be changed at any time by the user, this is described in more detail in Section 8.5.

Additionally, FunctionDefinitionAST nodes are enabled. KDevelop however has the same node for free functions and methods. Therefore four colors are assigned to this node, light green for public methods, light orange for protected methods, pink for private methods and light blue for free functions. The user can not only easily change the color of a construct, but he also can control which structures will actually show up in the view. This is described in more detail in Section 8.5.

As presented in Table 1 several kinds of AST nodes have the same color. By default we configured control structures, i.e. if statements and switch statements, to have the same colors. Similar, loop structures, i.e. for statements, do statements and while statements have the same color too.

Figure 14.: kdeui/kactionclasses.cpp results.

8.2.3 *Editor interaction and performing transformations*

Once the developer has selected a file in the tree, the file impact view will render the colored bars for the queries that have hits in the file. Clicking files in the tree does not open the files for editing. This way the developer can quickly examine the impact on the file and go the next file without being distracted by the low level details of the actual code. As an example we take the file `kdeui/kactionclasses.cpp` from kdelibs, for which the file impact view is shown in Figure 14.

From this picture we can make the following observations. First of all this seems to be a source file with quite some functions, indicated by the many small light green blocks (public methods) which occur on the whole range, some pink blocks near the middle (private methods) and some orange blocks (protected functions) left from the middle. Most of them are relative small functions, taking in account the width. The white spaces near the beginning and the middle seem to indicate even more code as we also see query hits in those areas. Furthermore there are some bigger functions which contain consecutive and nested control structures (purple blocks) and loop structures (brown blocks). Finally we also see that there is a class defined in this file (green block left from the middle).

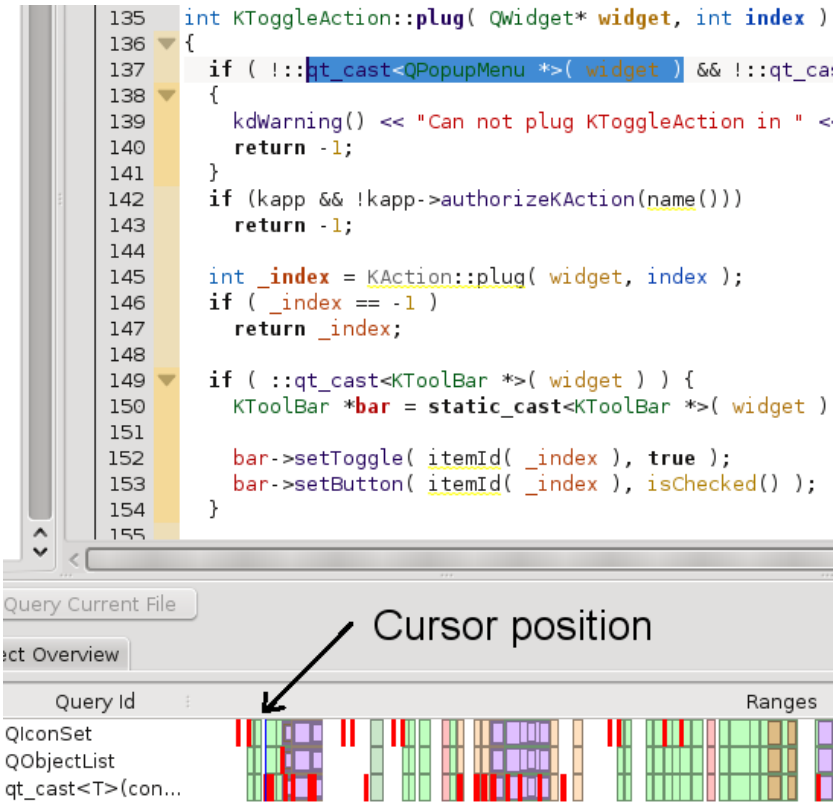
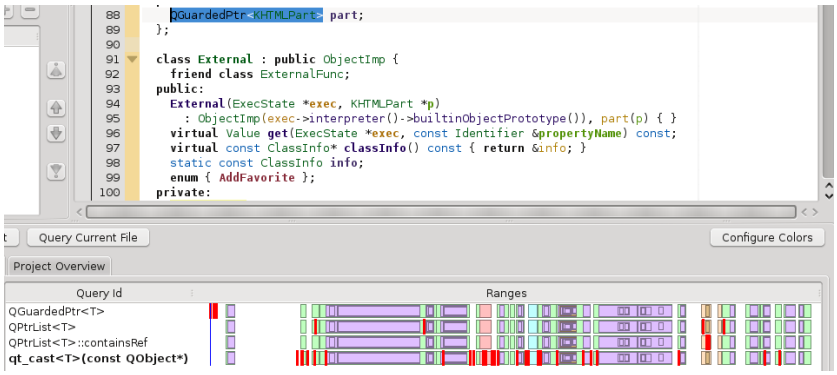


Figure 15.: kdeui/kactionclasses.cpp selected query hit.

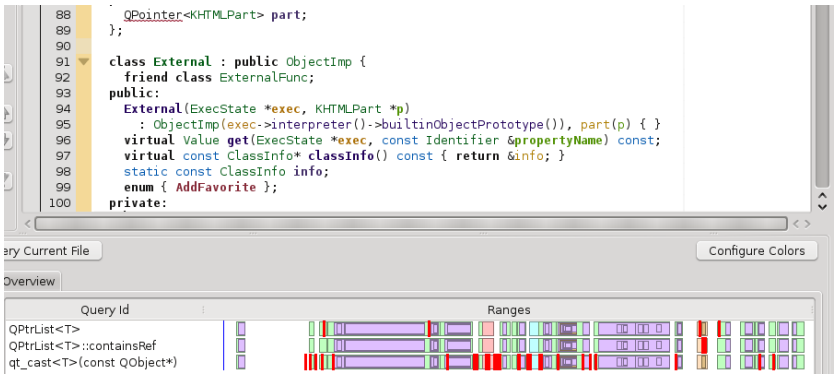
To start the actual porting for a specific file the developer clicks either on a location of interest to him in one of the colored bars or he clicks on one of the red patches. Both actions will open an editor for the source file and set the cursor at the position matching with location where the developer has clicked. In the latter case however, i.e. when the developer clicks a red patch, the editor will select the range in text which corresponds with the range of the hit that was clicked by the developer. For example, Figure 15 shows the selected text after clicking on the first red patch for the `qt_cast` query. Additionally, a thin blue marker in the colored bars indicates the current cursor position in the file. This marker is updated both when the user clicks in the colored bar and when he navigates through the source file in the editor.

Once the developer has clicked a red patch, he can directly start editing the code which will result in replacement of the selected text, which is as we showed the range of a query hit. However, in many cases a transformation will be defined for the query and performing the transformations automatically is the preferred way of doing the port. To this means the user can right click on the colored bar which will pop up a context menu giving him two options. When the right click was on a red patch he can choose to perform the transformation only for the particular hit. The second choice he has is to perform the transformation for all

hits of the query represented by the bar on which he clicked. Both actions will result in the editor being updated with the changes applied to the document.



(a) Before trasnformation.



(b) After transformation.

Figure 16.: File impact view before and after transformation.

When transformations are applied or when the source file was modified manually, the user can update the file impact view for the current source file. Figure 16 shows the impact view before (16a) the user choose to perform all `QuardedPointer` transformations and afterward (16b). In the latter we see that the hits for the `QuardedPointer` query have vanished from the view. The user can iterate this process of applying transformations and additionally doing manual modifications to the source file until it is completely ported.

8.2.4 *File impact view zooming*

On the one hand, conflicts between queries and thus transformations can occur as discussed in Section 7.5.1, on the other hand the horizontal space is limited, meaning that for large files it can happen that the red blocs of two hits for different queries interfere, while this is not really the case. An example of two possible interfering hits can be seen in Figure 17 which shows the result for the file `kdeui/kdockwidgets.cpp` from kdelibs. For this

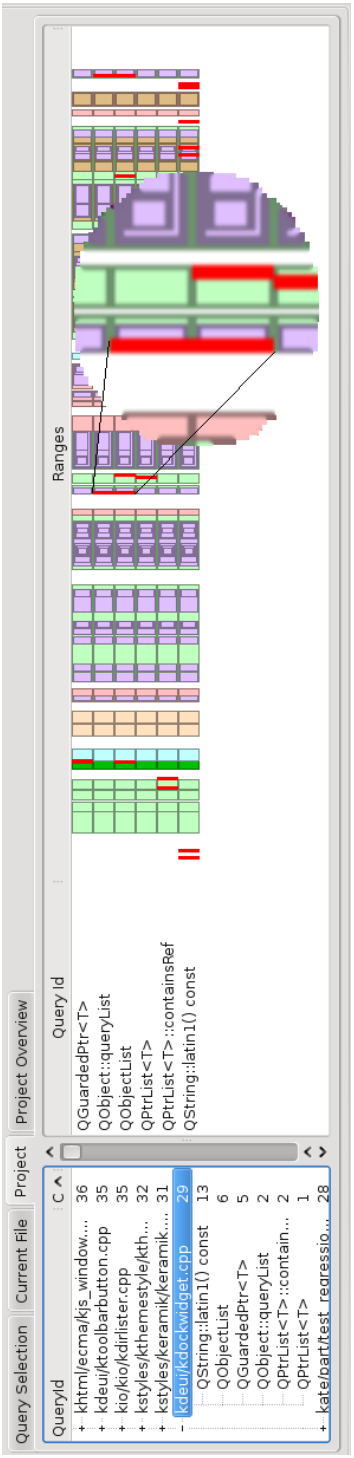


Figure 17.: Detail of file impact view showing possible conflict.

reason we added zooming functionality to the file impact view. The user can zoom the file impact view by holding the control key and rotating the scroll wheel of the mouse at the same time. Scrolling the mouse wheel up will zoom in, scrolling the mouse wheel down will zoom out.

Figure 18.: Detail of file impact view showing possible conflict.

Zooming in has several effects. This is visible in Figure 18 which shows the default view, i.e. a representation of the full content of the file, in the upper half and a zoomed in view of the same file in the lower half. The black lines indicates the corresponding areas in the source files in both views. The first effect is that rows will increase in height, up to a fixed maximum, when zooming in. This creates more space for nested constructs that might become visible due to zooming. The second effect is that, due to the fact that more space becomes available for a smaller amount of code, more details of the code appear in the view. For example, we see the white space between pink and purple block in the upper half of the image filled with various blocks in the corresponding area in the lower half of the image. Also, nested blocks, such as the brown block (loop structure) in the pink block (private method) become visible. Finally, it has become clear that the two hits, at the second and the third row, do not interfere with each other.

We implemented the zooming functionality as follows. The starting point for the algorithm is the mouse location in the colored bar. This is taken as the center point for the zooming action to prevent that the user has to move the mouse during zooming to keep track of the location of interest. Given the mouse position and the new zoom level, we calculate which part of the bar for the complete file is visible. Next, the block ranges are calculated on a normalized range for the complete document as described in Section 8.2.2. The normalized blocks are than translated to the visible rectangle. Blocks that fall outside the range result in invalid rectangles, ignored by the paint method and blocks that partly fall inside the visible rectangle result in blocks that are cut off appropriately. When zooming in, the colored bar represents only a part of the source file, meaning that more space is available for that part of the source. This, in turn

results in less blocks being filtered out due to the minimum size criterion.

8.3 USE CASE: API FEEDBACK AND REFACTORING ESTIMATION

Our plugin was also used for an upcoming refactoring in KDevelop, which reuses the Kate[5] libraries for its advanced editor features. After the release of KDE SC 4.4 it was decided by the Kate developers to deprecate the library for `SmartRanges`. This library offered the functionality to keep track of text snippets in a document, even when the location of these snippets change due to modification elsewhere in the document. A similar functionality will be offered by the new API. Because this functionality is quite widely used in KDevelop, the new design is of importance for the KDevelop developers. For this reason they used the plugin to answer the following questions:

- How to get an overview of which API is required.
- How to estimate the amount of work to port KDevelop to the new API.

The first question was of importance to be able to give feedback to the Kate developers. A query file was made, containing about 190 queries for the classes and functions of the `SmartRanges` library. Next the KDevPlatform and KDevelop code bases, together containing about 190 KLOC, where queried using this file. From the results a list of often used functions, therefore representing the most important functionality for KDevelop, was compiled and send to the Kate developers for adjustment of the new API.

Additionally, the developer compiled a list of functions which he expects to be complicated to port, such as functions which are no longer available in the new API. This specified list of functions was used to query the code base and the results where used in a similar way as described in Section 8.1.3 to estimate the porting effort.

8.4 USE CASE: DEPRECATED API TRACKING

Another use case, somewhat related to the Qt3 to Qt4 estimation and porting use case, is keeping track of deprecated API used in a project. Marking API as deprecated, using specific macros which will generate compiler warnings, is a common way in to notify users of a library that those specific classes or functions will get removed in a future version of the library. This same strategy is used in the kdepimlibs library, part of the KDE SC. The kdepimlibs library contains the core functionality for personal information management applications. The kdepim module of

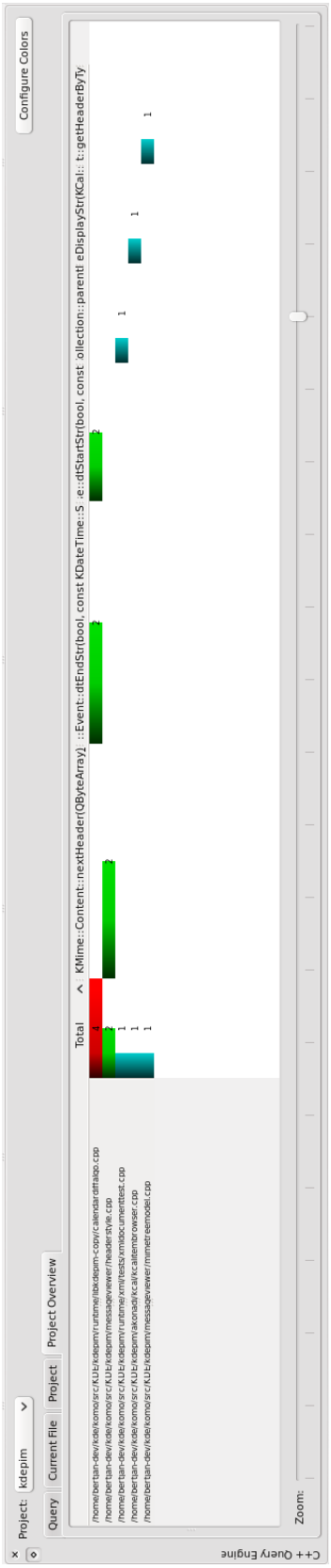


Figure 19.: Usage of deprecated kdepimlibs API in kdepim.

the KDE SC contains such applications and therefore depends on kdepimlibs.

For this use case we created a query file which contains queries for all methods that are marked as deprecated. This resulted in a file containing 47 queries for deprecated functions in 15 different classes[2]. This file was used to query the kdepim code base. We used trunk versions for both kdepimlibs and kdepim, i.e. the code bases which will be released with KDE SC 4.5.

Looking at the results, shown in Figure 19, a first observation is the surprisingly low number of total hits, given that kdepim has about 500 KLOC. As a reference we queried kdepim for uses of the `Akonadi::Collection` class. This resulted in 2043 uses in 373 different files. Given the high number of this class uses and the particular low number of uses of deprecated methods from this class, we can conclude that at least the code related to this is class is particular well maintained. In general, the code in kdepim seems to be well on track with its dependency kdepimlibs. This result can, to a large extend, be explained by the fact that kdepimlibs is mostly developed by a group of developers which also does a major part of the development in kdepim.

8.5 USE CASE: IDENTIFY WHICH PARTS OF A CLASS ARE AFFECTED

In order to estimate the impact of certain changes it is interesting to see where the code in a source file is affected. For example, changes in class definitions have a bigger impact than changes in the declaration of a private function of that same class, because the former will most likely also effect the "users" of that class. In order to different kinds of impact analysis, such as the one just described, we added a color configuration dialog, shown in Figure 20, which enables the user to enable or disable selection of AST nodes for visualization. For the enabled nodes he can configure the color which the blocks for the selected nodes will have. Changes in the configuration are directly applied to the view, so there is no need to query the file again.

For this particular use case we disable all AST nodes except the `ClassSpecifierAST` and the four `FunctionDefinitionAST` nodes. With this configuration the user gets a good overview of the structure of a source file. Figure 21 for example, shows the file kdeui/kdockwidget.cpp, which we saw in more detail in Figure 17. With the new configuration the global structure of the file becomes immediately clear. There are some small public methods (light green) in the beginning, followed by a class definition (green) a free function (blue) and some protected (orange) and

2 A copy of the file can be found at `http://bertjan.broeksemaatjes.nl/files/kdepimlibs-deprecated.pd`

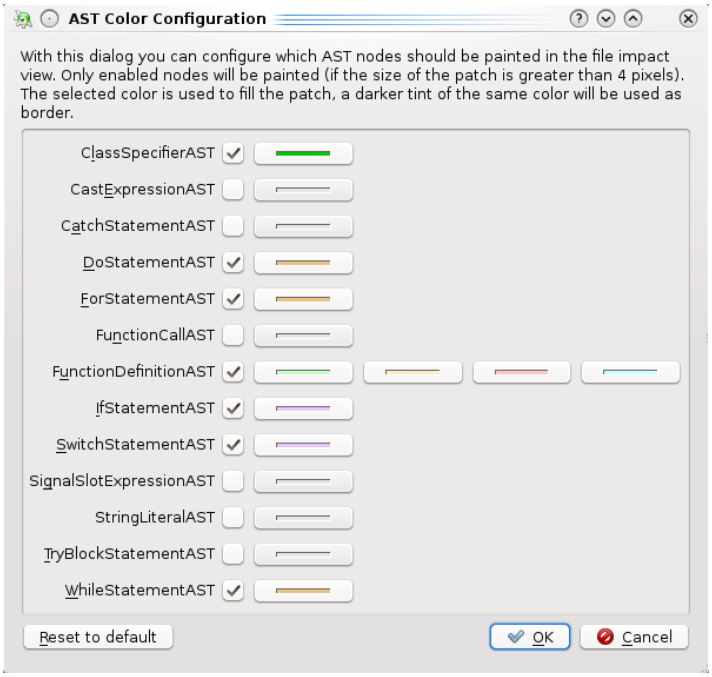


Figure 20.: The block color configuration.

private (red) methods. In the middle and more towards the end we see some bigger public methods. Looking at the distribution of red blocks, we can conclude that most of the changes occur in public method bodies.

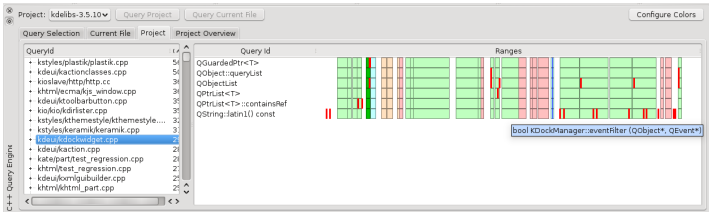


Figure 21.: File impact view configured to show less detail.

8.6 USE CASE: AFFECTED CODE COMPLEXITY

The final use case we present here is somewhat opposite to the previous use case. Another way to look at the complexity of the impact is by looking at code directly surrounding the hit for a particular query. More generally, developers want to have a quick idea of the complexity of a certain piece of code. Either because they need to modify it as part of a bug fix or feature implementation, or because they want to refactor certain parts of a source file to reduce the overall complexity. Deeply nested control structures and loops are often a cause that make code hard to understand for developers. Another code construct that

makes C++ code harder to understand is the `c-style` cast Sutter and Alexandrescu [34].

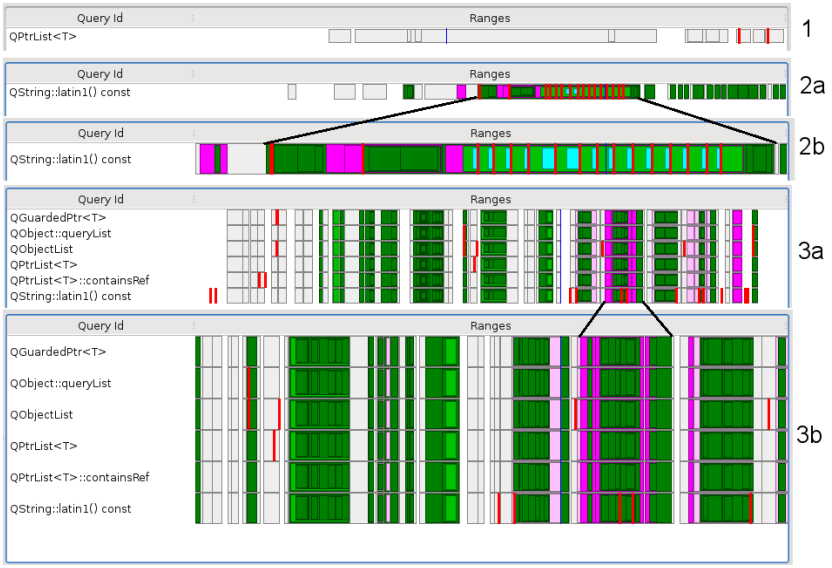


Figure 22.: Identifying complex pieces of code.

We adjust the configuration of the colors as follows. All coarse structures, i.e. class definitions and functions get a light tint to keep overview of the global structure of the file. Loop structures are configured to have different tints of purple and control structures are given different tints of green. Additionally we configured c-style casts to have a light blue color. Figure 22 shows the result when using this configuration for three files of kdelibs. The first file is a header in which only some classes are defined and therefore has no complexity. Next there are two files shown, both with a full file view and view zoomed in on a possibly complex piece of code.

In this chapter we discussed and demonstrated the visualizations that were added to the porting framework in order to support the users workflow. We demonstrated the project overview which shows the results of queries in a compact way using the table lens. It gives the user a quick overview on how the work is spread over a project with respect to the queries. Furthermore we demonstrated the file impact view which helps the user to quickly identify where a source file will be changed due to the porting. Finally, we demonstrated that the framework can be used for other tasks too by presenting the use case of code complexity analysis using the file impact view.

Part iii.

Evaluation and conclusion

9 CONCLUSIONS

Our main goal was to design and implement a general-purpose porting system. For proving its actual usefulness in practice, we implemented specific porting mechanisms, like queries and transformations, for porting C++ code from Qt3 to Qt4. The system should be able to handle real world code bases in terms of size and complexity of code. Other important requirements were ease of use, i.e. the framework should integrate well with the normal workflow of its users and customizability as different porting projects require different queries and transformations. Transformations should have as less impact as possible on the code. Finally, the porting system should provide means to the user to gain insight on the impact of the transformations on the source code before the changes where applied.

This is accomplished to a great extent with our extension for the KDevelop IDE. The extension gives the user the ability to specify queries for API that is subject to porting and perform these queries on a code base. Results of a query on a code base are presented on project level and on file level with visualizations that help estimating porting effort and impact of transformations on a file. In addition, transformations can be specified which enable the user to perform a large part of the porting work semi-automatically in a reliable way.

9.1 FACT EXTRACTION

To build our porting framework we needed a fact extraction front end for C++ for which we outlined the requirements in Section 2.2. The C++ front end that comes with KDevelop was able to meet most of these requirements. Being one of the main components in an IDE it necessarily is fault tolerant, i.e. it must be able to deal with code that is subject of constant changes. It parses large and complex code bases practically without any problems.

The main problem we have with this front end are the following. KDevelops C++ front end does not have an elaboration phase. This phase, which we find for example in the ELSA based frontends, makes explicit in the AST what is implicitly there according to the semantics of the language such as implicit casts and stack based destructor calls. Not only does this frontend lack an elaboration phase, the AST is highly simplified to make processing of it for the common tasks easier. In our opinion this has the

opposite effect on the tools depending on it such as the DUChain and our own tools. Unifying syntactical similar constructs results at various levels in checks for semantic differences which make the resulting code hard to understand and error prone. These are serious limitations which are not easy to solve as discussed in Boerboom and Janssen [20].

On the other hand, the DUChain is largely language independent. Currently there is a good frontend for PHP and frontends for Java and Ruby are under development. As discussed in Baxter et al. [19], language independence is an important criterion to build long term usable and scalable language tools. Although we build our tools for our specific needs, i.e. C++ porting, we believe that the properties of the KDevelop framework make it relative easy to extract major parts and reuse these for other language. Additionally, the C++ frontend has good support for Qt specific concepts such as the signal and slot mechanisms and properties. This makes the frontend and our tools an interesting platform to build new tools which check for problems related to these concepts.

9.2 QUERYING

For our porting framework we designed and implemented a lightweight C++ query engine on top of KDevelop's C++ front end, which we described in Chapter 5. This chapter also outlined the requirements for this query engine, ease of use (**QR1**), genericity (**QR2**) and customizeability (**QR3**).

EASE OF USE The query engine has a fairly simple and straight forward API which enables easy usage for custom use cases. However, for defining large amounts of queries, e.g. in the case of porting an API, this approach is not scalable. Ideally one would take a syntax which is as close to the source language as possible, such as proposed in Paul and Prakash [28]. This approach however, requires a complex processing stack for the query language alone and was therefore discarded. We wrapped the API in an XML format which is described in Appendix B. This has the advantage of being able to express the same queries as supported by the API without the need of recompiling in a language which is relatively easily to process. The drawback we found is that XML is not always that convenient to work with for the user. We therefore added a functionality to generate the correct XML snippets from the supported code constructs. Once the needed queries are specified they can be used on any C++ code base.

GENERICITY Although lightweight, our query engine still supports a wide range of programming constructs. The user can

query for uses of enumerations, classes, methods and free functions. In addition the engine has support for finding uses of template classes, uses of methods that take template arguments of the class it is member of and free template functions. This functionality is generic enough in practice to build up a set of queries for a Qt3 to Qt4 port which was the driving use case for our work.

CUSTOMIZABILITY Our goal was to provide means to specify restrictions on uses to be able to port different uses of the same construct in different ways. We added support for restrictions in two main areas. First, template based uses can be restricted on the template argument. Second, method and free function calls can be restricted by the values of the arguments. Especially the latter shows the power that is gained when using compiler techniques in comparison to scripts and regular expression based approaches. In Chapter 7 we demonstrated use cases that are supported by this functionality. These are only some of the many supported that require argument based porting.

PERFORMANCE The performance of the query engine is for the largest part determined by KDevelop's C++ front end. Although the query engine does some additional processing after it retrieved the results from the DUChain we did not notice significant difference in speed between performing one query or fifty queries. Performing twenty queries on the kdelibs 3.5 code base takes about thirthysix minutes on a 1.8 GHz dual core machine with 2 GB of ram. This is slightly faster than a compilation of the code base on the same machine. The problem with KDevelop's front end is that it does not store the AST. This has no effect when the user selected multpile queries, the source is parsed and then the same AST is used to perform all selected queries. However, when the same set of queries or a different query is performed after the previous run, all files need to be parsed and analyzed again. We believe that optimizations in various areas can be made to improve the performance of consecutive query runs on a project. Currently we worked around this problem by also offering the ability to update query results on a file basis, which can be done real time.

9.3 CODE TRANSFORMATION

Our goal was to design and implement a transformation engine that only changes affected code (**TR1**) and keeps the amount of changed code as small as possible (**TR2**). The transformation engine we designed reach both these goals by operating only on the results of the query engine and giving the freedom to the

user to change the whole range of the result or only specified parts. Because the transformation engine directly operates on ranges in the source text, i.e. there is no pretty printing of the complete source file involved in the process, white space changes and comment changes are reduced to the bare minimum. This approach makes the transformation also very fast, and enable the user to do the transformation real time.

The transformation engine is well suited for its driving use case Qt3 to Qt4 porting or in general, for porting between two similar APIs. A common property of these kind of ports is that the majority of the changes is local to the uses of affected API. This means that the majority of the transformations is fairly simple. The weak point of our approach is therefore that it does not work for transformations that need to change code around an use or need to make more structural changes to code. The latter would also require extension of the query engine to support finding code constructs not directly related to an use (e.g. loop structures and control structures). Another problem we did not solve is the problem of chained transformations which for example happen when the return type of a method changes.

9.4 VISUAL SUPPORT

As stated in **GR6**, our goal was to provide means to examine the impact of a port. The extension offers two visual components which implement this requirement on a project wide level and on a source file level. Use cases of both views are described in Chapter 8.

PROJECT OVERVIEW The project overview presents the results of queries in a table lens view. This way results in many files can be presented in a clarifying way. The default configuration of the view gives a clear overview on how the results of the queries, thus the amount of work, is distributed over the files in a code base. In addition the view supports various other use cases such as keeping track of progress during long porting projects and finding the distribution of API usage in a particular code base.

FILE IMPACT VIEW The file impact view uses dense pixel techniques to visualize code structure in a horizontal way in limited space. Blocks with different colors are used to represent various constructs. Nested constructs result in nested blocks. Blocks that have width or height under a certain limit are not painted to avoid clutter. Red patches with a width that is proportional to the range of the query hit, represented by the patch, in the source file. We use a smooth zooming mechanism to navigate between a full file overview and a more detailed view of specific parts of

the file, showing more details of the source file around the zoom location. Finally, we made this view interactive by linking it to the editor in three ways. Firstly, it allows the user to move the cursor of the editor to a specific place by clicking on a location in the file impact view. Secondly, a blue marker on the file impact view shows the current location of the cursor in the document and is updated when the cursor is moved in the editor. Lastly, clicking the red patches in the file impact result in a selection of the corresponding range in the editor, which allows quick modification of a query result by the user.

9.5 FUTURE WORK

In this final section we present some ideas for future improvements of the porting extension we developed for the KDevelop IDE. The first improvement we suggest is to extend our tool with scripting support at specified extension points. A second improvement we suggest is a GUI for specifying queries and transformations. Finally we suggest various improvements to the visual support of our tool.

9.5.1 *Scripting support*

The query engine as well as the transformation engine where designed with a particular aim at finding and transforming API. As we have seen, this works quite well for finding all uses of a specified API item and transformations locally to these uses. However, for ports where the changes are less local in comparison to changes required for Qt3 to Qt4 ports, more context when specifying queries and changes to this context are required.

Due to the nature of our target language, C++, which is highly flexible in the programming styles it supports, we think it is interesting to extend our plugin with scripting support. The approach could be to define one or more extension points in both the query engine and the transformation engine which will enable the user to plugin a script which gets access to a specified point in the AST. Examples of these extension points are the moment when the query engine finds a hit for a query and the moment of transformation as an alternative for insert and replacement actions.

Adding such an extension point to the query engine would allow the user the define project specific scripts for the more generic queries. Such script could for example check if the use reported by the query engine is located in a specific context, e.g. in a constructor or in the body of a for loop. Using scripting in the transformation engine will give the user the power to make changes to the context of an use.

Although adding scripting support might look counter intuitive given our initial arguments against scripting stated in Section 1.4.1 there are some good reasons to add this functionality. First of all, because the scripts are called at well defined points, there is much more control over what the scripts actually do, i.e. the user knows that the script will only be executed for well defined locations in the code. This is in contrast with plain usage of scripts for which the user never can know exactly what code will trigger the script. Secondly, creating query engines and transformation engines that are able to handle with the full complexity of porting projects are very expensive to build as pointed out by Boerboom and Janssen [20] and Akers et al. [16]. This pays off on the long term only and is therefore a high barrier for small and middle sized companies. Scripting support enables adding complexity as needed while the initial implementation is relatively easy given the available scripting framework provided by Qt on which our plugin is based.

9.5.2 *Defining queries and transformations*

Because our query language is particularly aimed at API it is fairly easy for users to specify basic queries for a given API. In addition, the added functionality of extracting query XML snippets from code, gives the tool a relative low barrier to get started. However, working with XML can be quite cumbersome especially when the XML file gets very big. We also have no support for extracting or defining transformations other than writing the needed XML code.

For this reasons it would improve the usability of the tool when a GUI would be developed that helps the user specifying and managing the queries and related transformations for a given API or project. Such a GUI can help improve the usability in two ways. Firstly, it can help the user to get a better overview for which parts of the API which is subject to porting he already has specified queries and transformations by providing space efficient and sortable views on the XML document. Second, it can help the user by improving the discoverability of the query and transformation language by providing the possible options supported by the language based on the context while the user specifies a query or transformation.

9.5.3 *Visual improvements*

Although our visualizations are already quite helpful while examining impact of API changes on a code base, we found that various improvements could be made both on the project overview and the file impact view.

PROJECT OVERVIEW A first improvement on the project view would be the addition of filtering. Currently it shows all results of all queries in all files. To support more fine grained analysis it would be needed to add the ability to filter out either parts of the project (i.e. certain directories or files) or the results of particular queries. The latter can be partly achieved currently but not without the need of requerying the code base. Another improvement would be to take in account extra information in table lens color calculation such as the number of conflicts in the file for the query, a user configured weight for each query or build and impact cost of a source file as described in Telea and Voinea [37]. This would enable even more precise estimation of the porting effort for a given code base.

FILE IMPACT VIEW Currently, the limited space approach for the file impact view results in some cases that red patches for different queries, indicating query result ranges in the file, are aligned even when the ranges do not overlap. This especially happens when the file represented by the view is very large. This problem is partially solved by adding the possibility to zoom in the view. However, for a quick indication of conflicts additional user interaction should not be needed. The view could therefore be improved to mark conflicting results in a different way than non-conflicting results.

As described in Chapter 8, one of the use cases of this view is identifying complex code parts in source files. This use case could be enhanced by adding the possibility to only visualize parts that adhere to specified conditions. Currently, displaying of code constructs depends on enabled syntax constructs and available space. However, in most cases this results in visualizations for complex as well as non complex parts of a source file. This could be enhanced by adding support for certain additional conditions such as minimum nesting depth and a required hit for a specified query. This would result in significant less constructs that get visualized and therefore it would become very easy to identify complex parts in a source file.

As a final remark we point out that all code that was written to implement the ideas presented in this thesis is available on a public git repository at:

`http://www.gitorious.org/kdevcpptools/kdevcpptools`

Part iv.

Appendix

A

QT3 TO QT4 PORTING EXAMPLE FILE

Listing 27: Queries and transforms for Qt3 to Qt4

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE porting-description SYSTEM
          "portingdatabase.dtd">
<porting-description>

  <queries>
    <!-- Function Queries -->
    <global-function-query
      uid="qt_cast&amp;lt;T&amp;gt;(const QObject*)"
      qid="qt_cast">
      <template-argument />
      <argument type="const QObject *" />
    </global-function-query>

    <!-- Enum Queries -->
    <enum-query uid="QButton::ToggleState"
                qid="QButton::ToggleState" />
    <enum-query uid="QIODevice::Offset"
                qid="QIODevice::Offset" />
    <!-- We can do enum values too -->
    <enum-query uid="QButton::Off"
                qid="QButton::Off" />

    <!-- Class function queries -->
    <!-- QString queries -->
    <class-function-query
      uid="QString(const std::string&amp;amp;)"
      qid="QString::QString">
      <argument type="const std::string&amp;amp;" />
    </class-function-query>

    <class-function-query
      uid="QString::operator=(const std::string&amp;amp;)"
      qid="QString::operator=">
      <argument type="const std::string&amp;amp;" />
    </class-function-query>

    <class-function-query
      uid="QString::ascii() const"
      qid="QString::ascii" const="true" />
    <class-function-query
      uid="QString::latin1() const"
      qid="QString::latin1" const="true" />
```

```
<!-- QObject queries -->
<!-- QObject::child(const char *objName,
                    const char *inheritsClass,
                    bool recursiveSearch)
-->
<class-function-query
  uid="QObject::child (all)"
  qid="QObject::child">
  <argument type="const char *" />
  <argument type="const char *" />
  <argument type="bool" />
</class-function-query>

<class-function-query
  uid="QObject::child (non recursive)"
  qid="QObject::child">
  <argument type="const char *" />
  <argument type="const char *" />
  <argument type="bool" >
    <restriction kind="QidRestriction"
                 value="false|FALSE|0" />
  </argument>
</class-function-query>

<class-function-query
  uid="QObject::queryList"
  qid="QObject::queryList"
  const="true">
  <argument type="const char *" />
  <argument type="const char *" />
  <argument type="bool" />
  <argument type="bool" />
</class-function-query>

<class-function-query
  uid="QObject::queryList (non recursive)"
  qid="QObject::queryList"
  const="true">
  <argument type="const char *" />
  <argument type="const char *" />
  <argument type="bool" />
  <argument type="bool">
    <restriction kind="LiteralRestriction"
                 value="false|FALSE|0" />
  </argument>
</class-function-query>

<!-- Class Queries -->

<!--
  - From Qt4 documentation:
  - classes that have been renamed in Qt 4. If
```

```
    - you compile your applications with QT3_SUPPORT
    - defined, the old names will be available.
    -
    - Whenever you see an occurrence of the name on
    - the left, you can safely replace it with the
    - Qt 4 equivalent in your program. The qt3to4
    - tool performs the conversion automatically.
    -->
  <class-query uid="QIconSet" qid="QIconSet" />

  <class-query uid="QWMatrix" qid="QWMatrix" />

  <class-query uid="QGuardedPtr&lt;T&gt;"
               qid="QGuardedPtr">
    <!--
      - This class has one template argument. We
      - need to specify it, otherwise the query
      - engine is not able to find the declaration.
      -->
    <template-argument />
  </class-query>

</queries>

<transformations>
  <!-- Function Queries -->
  <transform
    queryId="qt_cast&lt;T&gt;(const QObject*)">
    <rule>
      <replace-action item="FunctionId">
        qobject_cast
      </replace-action>
    </rule>
  </transform>

  <!-- Enum Queries -->
  <transform queryId="QButton::ToggleState">
    <rule>
      <replace-action item="All">
        QCheckBox::ToggleState
      </replace-action>
    </rule>
  </transform>

  <transform queryId="QIODevice::Offset">
    <rule>
      <replace-action item="All">
        qlonglong
      </replace-action>
    </rule>
  </transform>

  <transform queryId="QButton::Off">
```

```
    <rule>
      <replace-action item="All">
        QCheckBox::Off
      </replace-action>
    </rule>
  </transform>

  <!-- Class function queries -->
  <!-- QString transforms -->
  <transform
    queryId="QString(const std::string&amp;amp;)">
    <rule>
      <if>
        <condition
          property="ImplicitCtorCall"
          expected-value="true" />
        <insert-action item="ObjectId"
                       location="After">
          = QString::fromStdString
        </insert-action>
      </if>
      <else>
        <insert-action item="MemberId"
                       location="After">
          ::fromStdString
        </insert-action>
      </else>
    </rule>
  </transform>

  <transform
    queryId="QString::operator=(const std::string&amp;amp;)">
    <rule>
      <replace-action item="Arg[0]">
        QString::fromStdString(${Arg[0]})
      </replace-action>
    </rule>
  </transform>

  <transform queryId="QString::ascii() const">
    <rule>
      <replace-action item="FunctionId">
        toAscii
      </replace-action>
    </rule>
  </transform>

  <transform queryId="QString::latin1() const">
    <rule>
      <replace-action item="FunctionId">
        toLatin1
      </replace-action>
    </rule>
```

```
</transform>

<!-- QObject transforms -->
<transform queryId="QObject::child (all)">
  <rule>
    <!--
      - foo->child("objName")
      - becomes:
      - qFindChild<QObject *>(foo, "objName")
      -->
    <if>
      <condition property="ArgCount"
                 expected-value="1" />
      <replace-action item="All">
        qFindChild&lt;QObject *&gt;(${ObjectId},
                                    ${Arg[0]})
      </replace-action>
    </if>
  </rule>
  <rule>
    <!--
      - foo->child("objName", "Class")
      - becomes:
      - qFindChild<Class *>(foo, "objName")
      -->
    <if>
      <condition property="ArgCount"
                 expected-value="2" />
      <replace-action item="All">
        qFindChild&lt;${literalVal(Arg[1])} *&gt;
          (${ObjectId}, ${Arg[0]})
      </replace-action>
    </if>
  </rule>
</transform>

<transform
  queryId="QObject::child (non recursive)">
  <!--
    - foo->child("objName", "Class", FALSE)
    - becomes:
    - KGlobal::findDirectChild<Class *>(foo, "objName")
    -->
  <rule>
    <if>
      <condition property="ArgCount"
                 expected-value="3" />
      <replace-action item="All">
        KGlobal::findDirectChild
          &lt;${literalVal(Arg[1])} *&gt;
          (${ObjectId}, ${Arg[0]})
      </replace-action>
    </if>
```

```
  </rule>
</transform>

<transform queryId="QObject::queryList">
  <!--
    - foo->queryList("Class")
    - becomes:
    - qFindChildren<Class *>(foo)
    -->
  <rule>
    <if>
      <condition property="ImplicitOnThis"
                 expected-value="false" />
      <condition property="ArgCount"
                 expected-value="1" />
      <replace-action item="All">
        qFindChildren&lt;${literalVal(Arg[0])} *&gt;
          (${ObjectId})
      </replace-action>
    </if>
  </rule>

  <!--
    - queryList("Class")
    - becomes:
    - qFindChildren<Class *>(this)
    -->
  <rule>
    <if>
      <condition property="ImplicitOnThis"
                 expected-value="true" />
      <condition property="ArgCount"
                 expected-value="1" />
      <replace-action item="All">
        qFindChildren
          &lt;${literalVal(Arg[0])} *&gt;(this)
      </replace-action>
    </if>
  </rule>

  <!--
    - foo->queryList("Class", "objName")
    - becomes:
    - qFindChildren<Class *>(foo, "objName")
    -->
  <rule>
    <if>
      <condition property="ImplicitOnThis"
                 expected-value="false" />
      <condition property="ArgCount"
                 expected-value="2" />
      <replace-action item="All">
        qFindChildren&lt
```

```
          ;${literalVal(Arg[0])} *&gt;
          (${ObjectId}, ${Arg[1]})
      </replace-action>
    </if>
  </rule>

  <!--
    - foo->queryList("Class", "objName")
    - becomes:
    - qFindChildren<Class *>(foo, "objName")
    -->
  <rule>
    <if>
      <condition property="ImplicitOnThis"
                 expected-value="true" />
      <condition property="ArgCount"
                 expected-value="2" />
      <replace-action item="All">
        qFindChildren
          &lt;${literalVal(Arg[0])} *&gt;
          (this, ${Arg[1]})
      </replace-action>
    </if>
  </rule>
</transform>

<transform
  queryId="QObject::queryList (non recursive)">
  <rule>
    <replace-action item="All">
       KGlobal::findDirectChildren
         &lt;${literalVal(Arg[0])} *&gt;
         (${ObjectId}, ${Arg[1]})
    </replace-action>
  </rule>
</transform>

<!-- Class Queries -->
<transform queryId="QIconSet">
  <rule>
    <replace-action item="All">
      QIcon
    </replace-action>
  </rule>
</transform>

<transform queryId="QWMatrix">
  <rule>
    <replace-action item="All">
      QMatrix
    </replace-action>
  </rule>
</transform>
```

```
    <transform queryId="QGuardedPtr&lt;T&gt;">
      <rule>
        <replace-action item="TypeId">
          QPointer
        </replace-action>
      </rule>
    </transform>

  </transformations>

</porting-description>
```

B

PORTING FILE XML DTD

Listing 28: DTD for porting XML files

```
<!--
  Copyright (c) 2009, 2010 Bertjan Broeksema
  Copyright (c) 2010 Milian Wolff

  This file describes the XML format used for
  defining queries and transformations for the
  CPPPortingDatabase.

  This format is identified using the SYSTEM
  identifier "portingdatabase.dtd"

  Files using this format should include a DOCTYPE
  declaration like this:

  <!DOCTYPE porting-description
    SYSTEM "portingdatabase.dtd">

  It is possible to use xmllint which comes with
  XML Library libxml2:

  xmllint \
    **noout \
    **dtdvalid portingdatabase.dtd \
    yourdatabase.pd

  (Replace '*' with '-'. XML does not allow two '-'
   in comments)

  NOTE: some of the code in this DTD is from
        language.dtd of the Kate project.
-->


<!-- Boolean type
  Attributes that are of type boolean allow the
  following values:
  true, TRUE and 1 all meaning true,
  false, FALSE and 0 all meaning false.

  It is encouraged to use true and false
  instead of the alternatives.
-->
<!ENTITY % boolean "true|false|TRUE|FALSE|0|1">
```

```
<!-- Restriction kind type -->
<!ENTITY % restriction-kind
  "QidRestriction|LiteralRestriction">

<!-- Condition property type -->
<!ENTITY % condition-property
  "ImplicitCtorCall|ImplicitOnThis|ArgCount">

<!-- replace-action item type -->
<!ENTITY % action-item
  "MemberId|ObjectId|FunctionId|TypeId|Accessor|All" >

<!-- insert-action location type -->
<!ENTITY % insert-action-location "After|Before">

<!-- porting-description specification
  TODO: add attribs like name, version, author,
        license, ...
  TODO: add elements like description, ...
-->
<!ELEMENT porting-description
  (queries, transformations?)>

<!-- queries -->
<!ELEMENT queries
  (class-query|class-function-query
  |global-function-query|enum-query)*>

<!-- class-query specification
  uid: Unique descriptive name for this query,
       e.g.: Foo::Bar<T>
  qid: Qualified Identifier that should be
       matched by this class-query,
       e.g.: Foo::Bar
-->
<!ELEMENT class-query (template-argument)*>
<!ATTLIST class-query
  uid      CDATA  #REQUIRED
  qid      CDATA  #REQUIRED
>

<!-- class-function-query specification
  uid:   Unique descriptive name for this query,
         e.g.: Foo::Bar<T>::methodName(someType T)
  qid:   Qualified Identifier that should be
         matched by this class-query, e.g.:
         Foo::Bar::methodName
  const: Whether only const or non-const methods
         should be matched. [boolean, optional,
         default: both will be matched]
-->
<!ELEMENT class-function-query
  (template-argument|argument)*>
```

```
<!ATTLIST class-function-query
  uid      CDATA  #REQUIRED
  qid      CDATA  #REQUIRED
  const    (%boolean;) #IMPLIED
>

<!-- global-function-query specification
  uid: Unique descriptive name for this query,
       e.g.: NameSpace::funcName(t1 arg1, t2 arg2)
  qid: Qualified Identifier that should be
       matched by this class-query, e.g.:
       NameSpace::funcName
-->
<!ELEMENT global-function-query
  (template-argument|argument)*>
<!ATTLIST global-function-query
  uid      CDATA  #REQUIRED
  qid      CDATA  #REQUIRED
>

<!-- template-argument specification
  restriction: A typename that is matched against
               the instantiations and used for
               filtering them. [string, optional,
               default: empty]
-->
<!ELEMENT template-argument EMPTY>
<!ATTLIST template-argument
  restriction CDATA #IMPLIED
>

<!-- template-argument specification
  type: A typename that is matched against the
        declarations and used for filtering
        them. [string, optional, default: empty]
-->
<!ELEMENT argument (restriction)*>
<!ATTLIST argument
  type CDATA #IMPLIED
>

<!-- restriction specification
  kind:  The kind of restriction you want to
         impose (see top of this DTD for
         list of valid restriction kinds)
  value: The value the restriction should
         fulfil. How the matching is done
         depends on the restriction kind.
-->
<!ELEMENT restriction EMPTY>
<!ATTLIST restriction
  kind  (%restriction-kind;) #REQUIRED
  value CDATA                #REQUIRED
```

```
>

<!-- enum-query specification
  uid: Unique descriptive name for this query,
       e.g.: Foo::Bar<T>
  qid: Qualified Identifier that should be
       matched by this class-query,
       e.g.: Foo::Bar
-->
<!ELEMENT enum-query EMPTY>
<!ATTLIST enum-query
  uid      CDATA  #REQUIRED
  qid      CDATA  #REQUIRED
>

<!-- transformations specification
-->
<!ELEMENT transformations (transform)*>

<!-- transform specification
  queryId: the uid of the query you want to transform
-->
<!ELEMENT transform (rule)+>
<!ATTLIST transform
  queryId CDATA #REQUIRED
>

<!-- rule specification
-->
<!ELEMENT rule ((if, else*) |
                (replace-action|insert-action)*)>

<!-- if specification
-->
<!ELEMENT if (condition+,
              (replace-action|insert-action)+)>

<!-- else specification
-->
<!ELEMENT else (replace-action|insert-action)+>

<!-- condition specification
  property:       The property you want to match.
                  (see top of this DTD for list
                  of valid properties)
  expected-value: The value that this property
                  should have.
-->
<!ELEMENT condition EMPTY>
<!ATTLIST condition
  property       (%condition-property;) #REQUIRED
  expected-value CDATA                  #REQUIRED
>
```

```
<!-- insert-action specification
  location: location you want to insert the CDATA
            of this element at. (see top of this
            DTD for list of valid locations).
  item:     The item relative to which location
            is computed.
-->
<!ELEMENT insert-action (#PCDATA)>
<!ATTLIST insert-action
  location (%insert-action-location;) #REQUIRED
  item (%action-item;) #REQUIRED
>

<!-- replace-action specification
  item: The item you want to replace with the
        CDATA of this element. (see top of
        this DTD for list of valid items).
-->
<!ELEMENT replace-action (#PCDATA)>
<!ATTLIST replace-action
  item (%action-item;) #REQUIRED
>
```

BIBLIOGRAPHY

[1] Kde - an international technology team dedicated to creating a free and user-friendly computing experience, 2010. URL http://www.kde.org.

[2] Eclipse cdt - c/c++ development tooling, 2010. URL http://www.eclipse.org/cdt/.

[3] Eclipse - an extensible development platform, 2010. URL http://www.eclipse.org.

[4] Emacs - a customizable text editor and lisp interpreter, 2010. URL http://www.gnu.org/software/emacs.

[5] Kate - an advanced mdi text editor, 2010. URL http://kate-editor.org.

[6] Kdab - the qt experts. platform independent software solutions. URL http://www.kdab.com.

[7] Kdevelop - a free, open source ide, 2010. URL http://www.kdevelop.org.

[8] Mfc - the microsoft foundation classes, 2010. URL http://msdn.microsoft.com/en-us/library/d06h2x6e(VS.71).aspx.

[9] Motif - an industy standard graphical user interface, 2010. URL http://www.opengroup.org/motif/.

[10] Perl - a scripting language with good support for regular expressions, 2010. URL http://www.perl.org.

[11] Qt - cross-platform application and ui framework, 2010. URL http://qt.nokia.com.

[12] Qtcreator, 2010. URL http://qt.nokia.com/products/appdev/developer-tools/developer-tools.

[13] Sloccount - a aet of tools for counting physical source lines of code, 2010. URL http://www.dwheeler.com/sloccount/.

[14] Xrefactory - a c/c++ refactoring browser for emacs and xemacs, 2010. URL http://www.xref.sk/xrefactory/main.html.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[16] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Reengineering c++ component models via automatic program transformation. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, Washington, DC, USA, 2005.

[17] Robert Anisko, Valentin David, and Clément Vasseur. Transformers: a c++ program transformation framework. Technical Report 0310, EPITA/LRDE, 2003.

[18] Rudolf Ferenc Arp, Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting facts with columbus from c++ code. In *In Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004*, pages 4–8. IEEE Computer Society, 2004.

[19] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.

[20] F.J.A. Boerboom and A.A.M.G. Janssen. Fact extraction, querying and visualization of large c++ code bases. Master's thesis, Technische Universiteit Eindhoven, 2006.

[21] A. Van Deursen, J. Heering, H. A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: a component-based language development environment. pages 365–370. Springer-Verlag, 2001.

[22] Stephan Diehl. *Software Visualization. Visualizing the Structure, Behaviour, and Evolution of Software.* 2007.

[23] S.C. Eick, J.L. Steffen, and E.E. Sumner Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18:957–968, 1992. ISSN 0098-5589.

[24] R. Ferenc, A. Beszé, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for c++. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002.

[25] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[26] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. 2005.

[27] Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset

for source code investigation. *Information Visualization, IEEE Symposium on*, 0:4, 2005. ISSN 1522-404x.

[28] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20:463–475, 1994.

[29] Peter Pirolli and Ramana Rao. Table lens as a tool for making sense of data. In *AVI '96: Proceedings of the workshop on Advanced visual interfaces*, 1996.

[30] Ramana Rao and Stuart K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1994.

[31] Olaf Spinczyk. The puma project, 2010. URL `http://ivs.cs.uni-magdeburg.de/~puma/`.

[32] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*.

[33] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.

[34] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards. 101 Rules, Guidelines, and best practices.* Addison-Wesley.

[35] A Telea and H Byelas. Querying large c and c++ code bases: the open approach. In *Colloquium and Festschrift at the occasion of the 60th birthday of Derrick Kourie (Computer Science)*. Windy Brow, 2008.

[36] Alexandru Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *EuroVis*, pages 51–58, 2006.

[37] Alexandru Telea and Lucian Voinea. A tool for optimizing the build performance of large software code bases. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, 2008.

[38] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, pages 216–238, 2003.