PortAssist: Visual Analysis for Porting Large Code Bases

Bertjan Broeksema[*]
IBM Center for Advanced Studies (CAS), France

Alexandru Telea[†]
University of Groningen, the Netherlands

Abstract

We present PortAssist, an interactive visual analysis tool that helps C/C++ developers to estimate the effort needed, and automate refactoring, during software porting. We use automatic static analysis to find code constructs to be refactored and visualize the refactoring impact at project, file, and construct level. The tool integrates software querying, rewriting, and visualization into a KDevelop IDE plugin, and has been used to refactor industry-size code bases.

1 Introduction

When porting large software code bases, developers must perform many small-scale changes. An effort estimate, showing how changes depend on each other, and automating such changes, are keys to effective porting. We present PortAssist, a KDevelop extension [4] which assists porting industry-size C/C++ code bases. We support the assessment of overall porting effort and deciding the porting order by a project-level overview of the required changes at file level (for quick effort estimate), and a file-level view of how changes impact source code and depend on each other (for fine-grained analysis), using dense pixel-based techniques. The workflow integrates static analysis, source-code-querying, and program rewriting technologies.

2 Tool overview

The architecture of PortAssist is described below (see also Fig. 1).

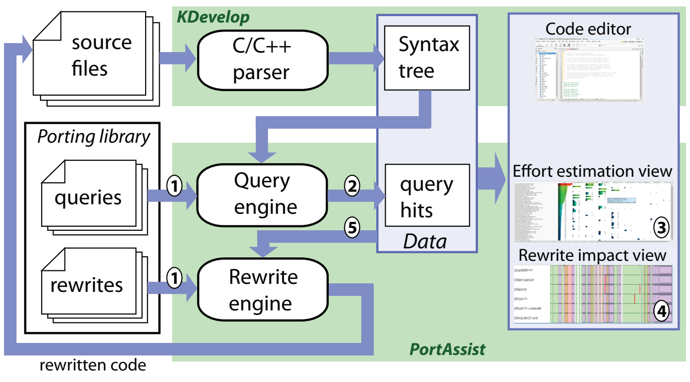


Figure 1: Architecture of PortAssist and its supported workflow

We first describe the constructs to be rewritten using a XML-based query pattern language on the abstract syntax tree (AST) of the input code. Constructs include usage of an API (functions or classes), usage of language features like copy constructors, default arguments, and templates, or any other C/C++ construct. Patterns are matched on the input code's AST extracted by KDevelop's built-in parser. Porting is specified similarly using program *transformations* which read an AST fragment (query hit) and output another AST fragment. Both queries and transformations are executed automatically, see examples at [1, 2].

[*]e-mail: bertjan.broeksema@fr.ibm.com; The work presented here was done while B. Broeksema was with KDAB Inc, Berlin, Germany

[†]e-mail: a.c.telea@rug.nl

The tool is used as follows (Fig. 1). We select a query-and-rewrite library specific to a given porting task (1) and run its queries (2). Query hits are shown in an effort estimation view to assess the overall porting effort and its spread over the code (3). We next select the most frequent hits and check how rewriting these hits affects the code (4). From these, we select hits which are safe to automatically rewrite (5). The process is repeated until porting is complete. We next detail the tool's views, using as example the porting of *kdelibs 3.5.10* (550 files, 750K lines) from version 3 of the Qt library (Qt3) to version 4 (Qt4). Queries and rewrite rules are based on the official Qt3-to-Qt4 porting guide [5].

2.1 Effort estimation view

This view uses a table-lens (Fig. 2, rows=files, columns=queries). Cells show the hit count in a file for a query. The leftmost column shows total hit count for a file. Sorting on this column, we see that most porting effort falls in 5..10% of all files. Brushing table cells highlights the actual hit code in KDevelop's editor, and shows that the main rewrites involve *qt_cast* (Qt typecasting macros), *QIconSet()* (constructing QIconSet objects), *QPtrList<T>* (list containers), and *QString::latin1* (string localization), *i.e.* columns 2 to 5, which have many large-value cells. To refine porting effort estimation, we define a quality level for each rewrite (how well the rewrite engine handles that construct) to color column headers (green=easy, red=difficult). In Fig. 2, we see that *QString::latin1* is frequent, but easily portable; however, *QPtrList<T>* affects many files, but is harder to rewrite.

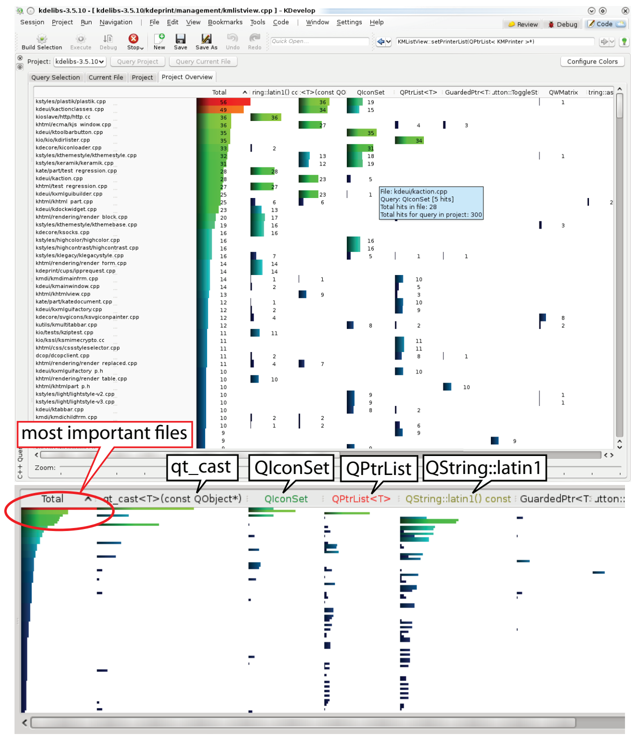
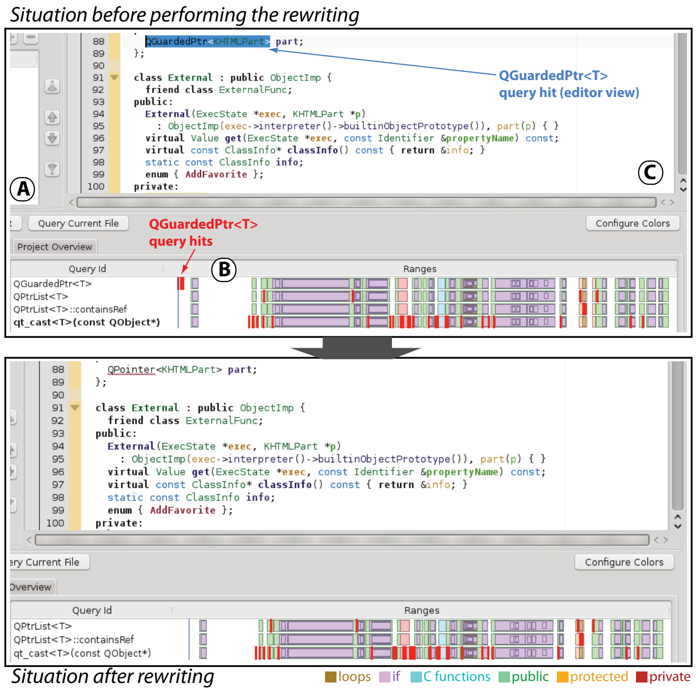

kdelibs-3.5.10 - [kdelibs-3.5.10/kdeprint/management/kmlistview.cpp] - KDevelop
Session Project Run Navigation File Edit View Bookmarks Tools Code Window Settings Help
Build Selection Execute Debug Stop New Save Save As Undo Redo
Project: kdelibs-3.5.10
Query Project
Query Current File
Configure Colors
Query Selection Current File Project Project Overview
Total
most important files
qt_cast
QIconSet
QPtrList
QString::latin1
Zoom:
Total
qt_cast<T>(const QObject*)
QIconSet
QPtrList<T>
QString::latin1() const
GuardedPtr<T::

Figure 2: Effort estimation view, sorted on total hits per file

2.2 Rewrite impact view

Developers want to see how rewriting changes their code before firing it off, *e.g.* code affected by several rewrite rules; and where rewrites occur in a file. The *rewrite impact view* addresses these questions (Fig. 3). Selecting a file (A) shows the query hits in that file, one horizontal bar for hits for a specific query, with hit locations in red (B). The *x* axis maps the file extent (left=first line, right=last line). Blocks represent AST constructs, scaled by size (lines of code), nested as they appear in the code, and colored by construct type (see color legend in Fig 3). Construct types not listed in the colormap and/or under a few pixels are not rendered to limit clutter. The view can be zoomed like a table lens. Vertical red bars indicate code containing *several* hits: For these, the user must choose the order of rewrite rules or handle them manually. The spread of red bars in the file shows where rewrites need to occur, *e.g.* in the preamble or main code range, or within function or class declarations. The view is linked with the code editor (C) by brushing and selection, allowing users to examine in detail the potential effects of a rewrite before actually performing it, as described next.



```
    QGuardedPtr<KHTMLPart> part;
  };

  class External : public ObjectImp {
    friend class ExternalFunc;
  public:
    External(ExecState *exec, KHTMLPart *p)
      : ObjectImp(exec->interpreter()->builtinObjectPrototype()), part(p) { }
    virtual Value get(ExecState *exec, const Identifier &propertyName) const;
    virtual const ClassInfo* classInfo() const { return &info; }
    static const ClassInfo info;
    enum { AddFavorite };
  private:
```

```
    QPointer<KHTMLPart> part;
  };

  class External : public ObjectImp {
    friend class ExternalFunc;
  public:
    External(ExecState *exec, KHTMLPart *p)
      : ObjectImp(exec->interpreter()->builtinObjectPrototype()), part(p) { }
    virtual Value get(ExecState *exec, const Identifier &propertyName) const;
    virtual const ClassInfo* classInfo() const { return &info; }
    static const ClassInfo info;
    enum { AddFavorite };
  private:
```

Figure 3: Potential rewrite inspecting (top). After the rewriting (bottom)

Several facts are visible in Fig. 3 top. Here, we selected a hit of the *QGuardedPtr<T>* query, which needs porting from Qt3 to its Qt4 syntax (*QPointer<T>*). All hits for this query occur outside function or class declarations (red block on white area at the left of topmost bar in Fig. 3 top), and that code is not affected by other queries (no red blocks beneath). Hence, we can safely use the rewriting engine for this rewrite. A right-button click on the query hit offers this possibility. Fig. 3 bottom shows the rewrite effect: The *QGuardedPtr<T>* query hits have disappeared, which has now three queries (colored bars) as compared to four before rewriting. The editor is updated too, since the rewrite changes the underlying source file.

Deciding whether to do an automatic rewrite, ordering of rewrites, and overall assessment of a rewrite's complexity depend on more than query hit count, hit locations, and hit overlaps. The *context* of a code fragment, *e.g.* surrounding code constructs and comments related to it, is important. To support better reasoning about a rewrite's impact, we add different color-mapped metrics to the impact view. One case which proved useful was to highlight changes in public method declarations as compared to changes of protected or private methods. This reflects the well-known fact that *interface* changes are more likely to propagate than implementation changes. Hence, rewriting a public method declaration will likely trigger more code changes in the very end, and should be done with great care. Separately, the quality of rewriting the *implementation* of a method depends on the complexity of the code around the hit location. For example, rewrites in deeply nested control/loop structures or C-style casts may make the code more unreadable than rewrites in simpler structures like sequences of assignment statements, since the former structures are already more complex than the latter even before rewriting [6]. Hence, rewrites inside complex structures should be done with greater care.

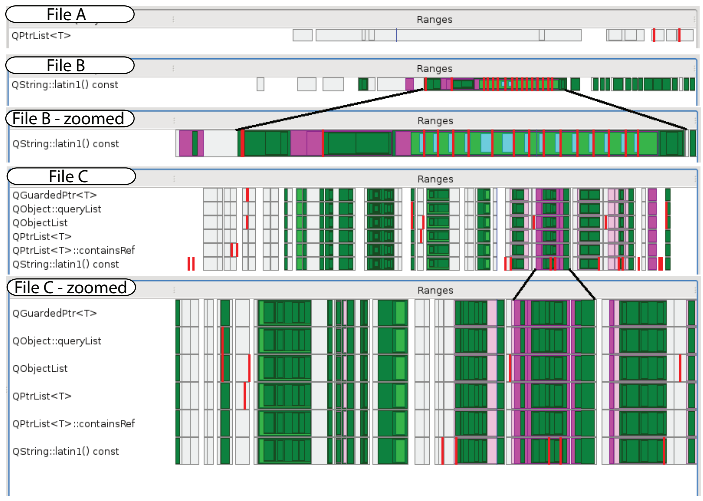


Figure 4: Rewrite impact view colored on rewriting complexity

We support such analyses by color-mapping complexity. Top-level structures, (*e.g.* class declarations and function bodies, get a light gray tint to show an overview of global structure. Different loop structures get different green tints depending on loop type (*e.g.* *for*, *do-while*). Control structures have different tints of purple depending on their type (*if*, *switch*, etc). C-style casts have a light blue color. Fig. 4 shows results for three files of *kdelib*. The first file (A) is a low-complexity, declaration-only, header (overall gray tint) and thus easily rewritable. For the next two files (B,C), we show a view of the whole file and a zoom-in of a complex part thereof. These files contain several deeply nested loop/control structures and C-style casts. In file B, the hits (*QString::latin1()* function calls) appear in a recurring pattern containing a *do-while* loop (light green) which ends with several C-cast statements (nested cyan block). The hits are the last block statements, *i.e.* loop control expressions, hence can be rewritten easily even though the loops themselves are complex. For file C, we show hits for six different queries. The first five queries have few, concentrated, hits as shown by the red bar locations (Fig. 4, file C, unzoomed view). The last hit, again for *QString::latin1()*, has more hits spread over a larger file area. Zooming in file C shows that most hits are outside complex structures, except for *QString::latin1()* hits (red bars inside green blocks). Hence, *QString::latin1()* rewrites in this file should be done carefully.

PortAssist was used for porting in an industrial setting, and is openly available as a KDevelop 4.0.1 extension [1], including query and rewrite rules, and demonstration videos [3].

References

[1] B. Broeksema. KDevelop C++ query and rewriting extension, 2010. http://www.gitorious.org/kdevcpptools/kdevcpptools.

[2] B. Broeksema. A visual tool-based approach to porting C++ code. MSc thesis, Dept. of Comp. Sci., Univ. of Groningen, the Netherlands, June 2010. http://www.cs.rug.nl/svcg/SoftVis/Refactor.

[3] B. Broeksema. Portassist video demonstration, 2011. `www.cs.rug.nl/svcg/SoftVis/Refactor`.

[4] KDevelop team. KDevelop IDE for C++, 2010. `www.kdevelop.org`.

[5] Nokia, Inc. Qt 3 to Qt 4 porting guide, 2011. `http://doc.qt.nokia.com/4.6/porting4.html`.

[6] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and best practices*. Addison-Wesley, 2005.