SUPPORTING CODE-DESIGN CONSISTENCY DURING SOFTWARE EVOLUTION

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Maen M. Hammad

May 13, 2010
Dissertation written by
Maen M. Hammad

B.S., Hashemite University, Jordan, 2002
M.S., Yarmouk University, Jordan, 2004
Ph.D., Kent State University, 2010

Approved by

Dr. Jonathan I. Maletic __, Chair, Doctoral Dissertation Committee
Dr. Robert A. Walker __, Members, Doctoral Dissertation Committee
Dr. Ruoming Jin ______
Dr. Michael L. Collard ___
Dr. Joseph D. Ortiz ______

Accepted by

Dr. Robert A. Walker __, Chair, Department of Computer Science
Dr. Timothy Moerland __, Dean, College of Arts and Sciences
TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................ VII
LIST OF TABLES ............................................................................................................ XI
ACKNOWLEDGEMENTS ............................................................................................... XIV

CHAPTER 1 INTRODUCTION ......................................................................................... 1
1.1 Maintaining Traceability between Design and Source Code ............................... 2
1.2 Goals of the Research ......................................................................................... 5
1.3 Contributions ....................................................................................................... 7
1.4 Organization ......................................................................................................... 8

CHAPTER 2 IDENTIFYING INCONSISTENCIES IN CODE-TO-DESIGN TRACEABILITY ................................................................. 9
2.1 Mapping Code Change to Design Change ............................................................. 10
2.2 Changes that Impact Design .............................................................................. 11
2.3 Changes that do Not Impact Design .................................................................... 12
2.4 Automatically Identifying Design Changes ......................................................... 14
   2.4.1 Design Change Identification ......................................................................... 18
   2.4.2 Classes and Methods ..................................................................................... 19
   2.4.3 Relationships .................................................................................................. 21
2.5 Evaluation ............................................................................................................ 26
   2.5.1 Design Changes in HippoDraw ...................................................................... 27
   2.5.2 The Study ..................................................................................................... 28
   2.5.3 Results ........................................................................................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6 Threats to Validity and Limitations</td>
<td>34</td>
</tr>
<tr>
<td>2.7 Related Work</td>
<td>35</td>
</tr>
<tr>
<td>2.7.1 Software Evolution</td>
<td>35</td>
</tr>
<tr>
<td>2.7.2 Design Evolution</td>
<td>37</td>
</tr>
<tr>
<td>2.7.3 Recovering Traceability Links</td>
<td>39</td>
</tr>
<tr>
<td>2.8 Summary</td>
<td>41</td>
</tr>
<tr>
<td><strong>CHAPTER 3 CHANGES IN CODE-TO-DESIGN CONSISTENCY DURING EVOLUTION</strong></td>
<td>43</td>
</tr>
<tr>
<td>3.1 Data Collection</td>
<td>45</td>
</tr>
<tr>
<td>3.2 Categorization of Commits</td>
<td>48</td>
</tr>
<tr>
<td>3.3 Design Changes and Design Stability</td>
<td>51</td>
</tr>
<tr>
<td>3.4 Characteristics of Commits</td>
<td>56</td>
</tr>
<tr>
<td>3.5 Bug Fix vs. Design Change</td>
<td>62</td>
</tr>
<tr>
<td>3.6 Commit Labeling</td>
<td>65</td>
</tr>
<tr>
<td>3.7 Threats to Validity</td>
<td>67</td>
</tr>
<tr>
<td>3.8 Related Work</td>
<td>67</td>
</tr>
<tr>
<td>3.9 Summary</td>
<td>69</td>
</tr>
<tr>
<td><strong>CHAPTER 4 MEASURING CLASS IMPORTANCE IN CODE-TO-DESIGN INCONSISTENCY</strong></td>
<td>70</td>
</tr>
<tr>
<td>4.1 Classes Impacted by Design Changes</td>
<td>71</td>
</tr>
<tr>
<td>4.2 Importance of Individual Classes</td>
<td>73</td>
</tr>
<tr>
<td>4.3 Importance of Multiple Classes</td>
<td>76</td>
</tr>
</tbody>
</table>
4.3.1 Itemset Mining ........................................................................... 79
4.3.2 Hierarchical Clustering .............................................................. 82
4.4 Evaluation..................................................................................... 87
  4.4.1 Itemset Mining ....................................................................... 87
  4.4.2 Clusters ............................................................................... 89
  4.4.3 Variability of Clusters ............................................................. 90
  4.4.4 Itemset Mining vs. Clustering ............................................... 92
  4.4.5 Threats to Validity ................................................................. 92
4.5 Related Work.............................................................................. 93
4.6 Summary .................................................................................. 96

CHAPTER 5 PRESERVING CODE-TO-DESIGN CONSISTENCY DURING
EVOLUTION .................................................................................. 98

5.1 Monitoring Code-Design Traceability .......................................... 98
  5.1.1 UML Completion Diagram .................................................... 100
  5.1.2 Using a Completion Diagram .............................................. 102
  5.1.3 Creating Completion Diagrams .......................................... 104
  5.1.4 Discussion ......................................................................... 109
5.2 Updating Design Documents ...................................................... 110
  5.2.1 The Method ........................................................................ 110
  5.2.2 Updating Design Database .................................................. 112
5.3 Related Work............................................................................ 114
  5.3.1 UML Visualization .............................................................. 114
LIST OF FIGURES

Figure 1—1 Code-to-Design traceability links are broken during code evolution........... 3

Figure 2—1 Example of code change that impacts design by adding the method
    writeToFile and a dependency relationship between classes PyFitsController and
    QtCut ........................................................................................................................................ 10

Figure 2—2. A code change that impacts the design due to the addition of a new
    association relationship between classes FitsNTuple and DataColumn....................... 12

Figure 2—3 Code changes from two revisions of domparser.cpp (KDE) that do not
    impact the design .............................................................................................................. 13

Figure 2—4 Three examples of code changes to method signatures that do not impact the
    design.................................................................................................................................. 14

Figure 2—5 Source code of swapping two numbers (A) and its srcML representation (B).
    Syntactic elements are marked as tags around the original (escaped) source code text.
    ........................................................................................................................................... 16

Figure 2—6. A srcDiff program fragment with selected srcML markup. The srcDiff tags
    are shown in bold.................................................................................................................. 17

Figure 2—7 The partial srcDiff of ColorBoxPointRep.h. A new method setBoxEdge
    was added ............................................................................................................................ 19

Figure 2—8 The partial srcDiff of VariableMesh.h. The new class VariableMesh has
    been added............................................................................................................................ 20
Figure 2—9  The partial srcDiff of RootController.h. The supertype Observer forms a new generalization ................................................................. 23

Figure 2—10  Partial srcDiff of DataView.cxx.  The method has a new declaration that uses PlotterBase, producing a potential new dependency ........................................... 23

Figure 2—11  Partial srcDiff of FitsNTuple.h. The class exists in both versions. The new data member uses type DataColumn forming a potential new association ............. 24

Figure 2—12  Partial diff output of code changes in the body of method DataView::prepareMarginRect. Two potential new dependencies have been added from class DataView to PlotterBase and to FontBase classes............................. 25

Figure 2—13  One of the 24 test problems given to human experts. The line differences are the relevant output of the diff utility. The UML class diagram is the pertinent parts of the system before the design change. Based on this information, the expert answered the questions at the bottom regarding changes to the design .................... 30

Figure 3—1  Part of code changes in revision 723866 on Kate. The change impact impacts design of Kate by adding/deleting three generalizations. ..................... 44

Figure 3—2  The log message of revision 502508 on Kate extracted from the Subversion repository .................................................................................................................. 45

Figure 3—3  The extracted XML representation of log message in Figure 3—2 ........... 46

Figure 3—4  Histograms of commits to (a)Kate, (b)KSpread over 3 year time period. For each month commits are categorized based on impact to design. In almost all months the number of commits with design changes is much less than the number of commits with no impact to design ................................................................. 49
Figure 3—5 Histograms of commits to (a)QuantLib and (b)wxWidgets over 3 year time period. For each month commits are categorized based on impact to design. In almost all months the number of commits with design changes is much less than the number of commits with no impact to design.

Figure 3—6 The distribution of commits into quartiles (x-axis values) based on the number of changed files per commit. The percentages of commits with design impact and with no impact based are shown. In general, commits with a design impact have a larger number of changes files.

Figure 3—7 The distribution of commits into quartiles (x-axis values) based on the number of changes (added/deleted) lines of code per commit. The percentages of commits with design impact and with no impact based are shown. Commits with a design impact consist of more changed lines.

Figure 4—1 Word cloud view for classes involved in design changes for (A) Kate and (B) KSpread. Size of the word (class) reflects the importance.

Figure 4—2 Example of code change with collaborative classes that impacted together by design changes.

Figure 4—3 Portion of dendrogram resulted by applying hierarchal clustering on commits of Kate.

Figure 5—1 A UML target diagram with the desired design.

Figure 5—2 A UML completion diagram. To complete the maintenance tasks the code should be changed by deleting the elements in red and adding the elements in gray.
Figure 5—3 Completion diagrams for visualizing the progress of a maintenance task for the target design in Figure 5—1. The starting completion diagram is shown in Figure 5—2. Each part shows the effect of code changes towards the completed task.

Figure 5—4 Visual description for the designTracer framework
LIST OF TABLES

Table 2-1  Design changes automatically identified in HippoDraw release 1.19.1 compared to release 1.18.1 by the tool srcTracer ................................................................. 28

Table 2-2 The comparisons of the results between tool and the three human subjects .... 33

Table 3-1 For each studied project, the directory the files came from, the time duration over a 3 year period, the total number of source files at the beginning of the time period, and the total studied commits (after filtering) .................................................. 47

Table 3-2 Percentages of commits with design impact and with no design impact for the four projects during the three year time period ......................................................... 48

Table 3-3 Distribution of commits based on their design impact category on Kate and KSpread ........................................................................................................................................ 53

Table 3-4 Distribution of commits based on their design impact category on QuantLib and wxWidgets ........................................................................................................................................ 53

Table 3-5 Distribution of Design changes occurred in May 20007 on Kate. The month has the largest number of design changes. Possible interpretations can be identified from detailed changes ........................................................................................................... 55

Table 3-6 For each project, the average size of the commit (number of files) and the average number of lines changed (i.e., added and deleted) ........................................... 57

Table 3-7 Summary of the max, min, and average number of add/deleted design elements by commits with design impact ................................................................................................. 61
Table 3-8 Grouping of commits based on design impact, bug fixes, and both............. 63
Table 3-9 An example of labeling commits that have design impact on Kate .............. 66
Table 4-1 Information about the extracted commits from Kate and KSpread................ 72
Table 4-2 Top 10 classes with highest importance on Kate and KSpread.................... 74
Table 4-3 Distribution of classes based on their importance for Kate and KSpread....... 75
Table 4-4 Results of the identified itemsets by Apriori for Kate (support=3) and KSpread (support=5) ........................................................................... 80
Table 4-5 Some identified itemsets (collaborative classes) from Kate by Apriori....... 81
Table 4-6 Information about identified clusters by hierarchal clustering for Kate and KSpread .................................................................................... 85
Table 4-7 Some Identified Clusters of commits by the hierarchal clustering on Kate..... 86
Table 4-8 Evaluation results for identified item sets for Kate and KSpread with different Support values .................................................................................................. 88
Table 4-9 Evaluation results for identified clusters for Kate and KSpread (low recall and high precision) ..................................................................................... 90
Table 4-10 Distribution of clusters based on the Variability value for Kate and KSpread ........................................................................................................ 92
Table 6-1 Evaluation of itemsets-Kate–Support=2.................................................... 123
Table 6-2 Evaluation of itemsets-Kate–Support=3..................................................... 123
Table 6-3 Evaluation of itemsets-Kate–Support=3..................................................... 123
Table 6-4 Evaluation of itemsets-KSpread–Support=4............................................. 124
Table 6-5 Evaluation of itemsets-KSpread–Support=5 .......................................... 124
Table 6-6 Evaluation of clustering-Kate ............................................................................. 124
Table 6-7 Evaluation of clustering-KSpread .......................................................................... 124
ACKNOWLEDGEMENTS

First of all I thank Kent State University and Computer Science Department which gave me the opportunity to continue my graduate study in computer science.

Words fail to express my deep gratefulness to my parents who did more than their best to raise, educate and support me. This dissertation is an outcome for their great efforts over long years.

I am grateful to all my colleges in the software engineering development laboratory (SDML) who helped me in different ways to accomplish my research and dissertation. I would like also to thank Dr. Michael Collard for his valuable contributions.

I am deeply grateful and thankful to my advisor Dr. Jonathan Maletic for his great support and outstanding help during my PhD study and research. I am deeply grateful for his great cooperation and efforts in order to accomplish this dissertation.

Finally, I thank my dissertation committee. I really appreciate their efforts and participations.

Maen M. Hammad

May 2010, Kent, Ohio
CHAPTER 1

Introduction

Software systems are composed of a variety of artifacts. Source code, instruction manuals, and design documents are examples of software artifacts. Source code and design documents are two critical artifacts of software systems. Design is specification of the solution [Chikofsky, Cross 1990] and is a high-level description of the implementation. Once the solution of the problem is specified (design), programmers implement this specification in programs (source code).

The UML (Unified Modeling Language) is widely used to represent high-level design models. It is “a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system” [Booch 1999]. Specifically in the UML Class Diagram the main parts of object-oriented design can be specified including what classes make up the system, the attributes and operations of those classes, and the relationships between the classes in the system. The Object Management Group (OMG) adopted UML as a standard in 1997 and it is now the de facto standard throughout the software engineering industry.

All software systems undergo maintenance and evolution activities. According to IEEE standard (1219-1998), software maintenance is the “process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment”. It is widely accepted that a large portion of the cost of a software system over its lifetime is spent on maintenance
activities [Schach 2005]. Therefore, a primary challenge of software engineering is to ease maintenance activities to reduce their cost over the lifetime of the system.

To achieve this goal, both design and code must be consistent at all points of the project’s lifetime, including maintenance. In order to understand the source code undergoing a maintenance activity, the corresponding design is investigated. It is often the case that code and design become inconsistent. This occurs because changes to the source code are not consistently made to the corresponding design, or the design artifacts were never created or were lost.

The process of design recovery from source code is called reverse engineering [Chikofsky, Cross 1990]. Existing source code is automatically analyzed to produce design documents, typically UML Class Diagrams. While these recovered design artifacts may be useful, manually constructed design documents also include a wealth of vital information such as meaningful diagram layout, annotations, and stereotypes. This type of meta-data is very difficult to derive or reconstruct via reverse engineering of the design from the source code. There are some automated methods to assist in this recovery but these approaches cannot recover valuable meta-data. Reverse engineering process can be avoided if the design is available and consistency with code at any point in the lifetime of the project.

1.1 Maintaining Traceability between Design and Source Code

During the initial stages of a software project there is often a great deal of energy and resources devoted to the creation of design documents. UML class diagrams are one of the most popular means of documenting and describing the design. A major reason for
the popularity of UML is the clear mapping between design element and source code. As such, initially the traceability between design-to-code and code-to-design is consistent and accurate. That is, the class diagram expresses the current state of the source code. Traceability links can be easily defined at this point in development, however little tool support exists for this task and rarely do traceability links exist explicitly in a useable manner.

![Diagram showing code-to-design traceability before and after code change](image)

**Figure 1—1 Code-to-Design traceability links are broken during code evolution**

During evolution, change occurs to the source code for many reasons (e.g., fixing a bug or adding a feature). This creates the serious problem of keeping the design artifacts in-line and current with the code. Figure 1—1 shows a visual description of the problem. The consistency of the traceability links from code-to-design is regularly broken during code changes, and as a result the consistency between code and design is lost and the design documents will soon decay without expensive and time-consuming manual upkeep.
Developers also lose the architecture knowledge of the system during evolution [Feilkas, Ratiu, Jurgens 2009]. This leads to incorrect implementation for the intended design. As a result, code-design traceability will be incorrectly established and code will be inconsistent with design. It is essential to keep developers on track toward correctly establishing code-to-design traceability during code change activities.

One solution to the code-design consistency problem is to reverse engineer the entire class diagram after a set of changes. However as mentioned previously, a large amount of valuable meta-information is lost. We feel that reverse engineering a complete design is unnecessary if some consistent design exists. An incremental analysis of the changes, in step with system evolution, should produce a much richer and more accurate design document. Another problem is the computational overhead needed to periodically run reverse engineering process.

There is a lack of efficient automated techniques to identify and maintain traceability and consistency between code and design during evolution. Most of the related work in the area of software evolution focuses either on code evolution or design evolution. In code evolution researchers study the evolutionary patterns in size of the code, including the number of lines (LOC), files, and functions whereas in design evolution the focus is on design differences, changes, and architectural evolution. Mens [Mens et al. 2005], [Mens, Demeyer 2008] states that “It is necessary to achieve co-evolution between different types of software artifacts or different representations of them” and stresses this as a specific challenge in software evolution with code-to-design as a major research problem.
1.2 Goals of the Research

The dissertation focuses on automatic analyses of code changes in software systems and understanding how these changes impact the design. Based on this understanding, a set of automated tools and approaches are developed to assist in keeping code and design consistent during software evolution.

To maintain consistency each change to the source code must be examined and comprehended to evaluate its impact on the design. The main idea behind the approach taken here is to construct automated methods to assess if a given code change impacts the design. This information can be used in a number of ways. It can be used to inform developers with the necessary adjustments to the design documents. Additionally, it could be used to reveal valuable information about the evolutionary history of the software. It could also be used to automatically maintain design documents and to monitor the progress of evolution.

The dissertation addresses the following main three program comprehension research questions:

- How does a code change impact the design?
- How does the design change during daily and incremental code changes?
- How to preserve the consistency of code-to-design traceability links during code changes?

The first and third research questions are addressed in Chapters 2 and 5. The second research question has been addressed in Chapters 3 and 4.
The first research question is key. Given a code change, does this change alter the design model of the software? An efficient automatic approach is needed to address this question. The existence of such an approach allows us to address the subsequent questions.

The second question is a direct extension of the first. It is an attempt to understand the impact of code changes that developers commit over time, on the design. Not all code changes will impact the design; therefore we classify code changes into two categories; code changes with a design impact and code changes with no impact on design. Studying the distribution of these categories over time is an interesting issue. The goals of the study are:

- Identifying periods of design stability
- Understanding the characteristics of each category and the benefits from this understanding, e.g., if the code change does not impact the design it may be a bug fix
- Measuring the importance of a particular class in the context of design evolution (code-design inconsistency).

The goal of the third question is to provide methods to support keeping code and design consistent (co-evolution) during code changes. Specifically this means to:

- Visually guide developers to the target design to avoid code-design inconsistencies and incorrectness.
- Automatically update and maintain design documents to keep them consistent with code.
1.3 Contributions

The dissertation work provides the following research contributions in the field of software evolution:

- A lightweight and efficient automated approach to identify inconsistencies in code-to-design traceability given a code change.
- A detailed empirical study to understand how code-to-design traceability is changed during evolution and the useful benefits and applications for this understanding.
- An approach to measure class importance based on its involvement in changes to design.
- Two methods to preserve the consistency of code-to-design traceability during software evolution:
  1. An environment to monitor and aid developers towards implementing the intended design.
  2. An automated prototype to update and maintain design documents during evolution.

This work adds to the knowledge of how large-scale software systems evolve over time and how source code changes impact the broader design of the system. This is critical to sustain the quality goals of a software system over time. Maintaining the quality of software impacts the daily life of almost everyone in today’s modern society.

The first research contribution (Chapter 2) is published in the 17\textsuperscript{th} IEEE International Conference on Program Comprehension (ICPC’09) [Hammad, Collard,
Maletic 2009]. The second research contribution (Chapter 3) is detailed in an invited submission to the Software Quality Journal (SQJ) [Hammad, Collard, Maletic 2010b] and it has been accepted for publication. The third contribution (Chapter 4) is partially presented and published in the 18th IEEE International Conference on Program Comprehension (ICPC’10) [Hammad, Collard, Maletic 2010a].

1.4 Organization

Each chapter of the dissertation addresses a specific research issue and corresponds to a paper that is published, under review, or being prepared for submission. CHAPTER 2 details the automated approach that identifies breaks in code-to-design traceability during evolution. CHAPTER 3 empirically investigates how code-to-design traceability is changed over daily and incremental code changes and the benefits gained from this understanding. CHAPTER 4 presents an approach to measure class importance in the context of design evolution. It is a direct application for identifying incremental design changes. CHAPTER 5 presents two methods used to support in preserving the consistency of code-to-design traceability during code changes. Conclusions and future work are presented in CHAPTER 6.
CHAPTER 2

Identifying Inconsistencies in Code-to-Design Traceability

This chapter specifically addresses the following program comprehension question. *Does a set of code changes impact the design?* By automatically answering this question we can then address how to ensure consistency of code-to-design traceability during code evolution. It also directly supports the comprehension of a code change.

Of course, not all changes to the code impact the design. For instance, changing the underlying implementation of a data structure or changing the condition of a loop typically does not change the design, while changing the relationship between two classes or adding new methods generally has a real impact on the design. So, given a set of code changes, it is a non-trivial task to determine if there needs to be a corresponding change to the design.

The approach does not rely on the existence of explicit traceability links between code and design. Nor does it actually require the existence of a design document (class diagram). All that is used to determine if there is a design change is the source code (in this case C++) and details about the change (i.e., the diff). However, for the question to be completely answered there should exist a version of the source code and design document that were consistent at some point in time. The change history of the source code is readily available from a version-control system such as CVS or Subversion.
The Chapter has been published in the 17th IEEE International Conference in Program Comprehension (ICPC’09) [Hammad, Collard, Maletic 2009] and it is organized as follows. Sections 2.2, 2.3, and 2.4 describe and distinguish between code changes that impact design and changes that do not. The approach and the automated tool are detailed in Section 2.4. The evaluation of the approach on an open source project is presented in Section 5. Threads to validity and related work follow.

```cpp
--- ../HippoDraw-1.18.1/python/PyFitsController.h
+++ ../HippoDraw-1.19.1/python/PyFitsController.h

class FitsController;
class FitsNTuple;
class PyDataSource;
+ class QtCut;

+ void writeToFile ( const DataSource * source,
+                   const std::string& filename,
+                   const std::vector<QtCut * >&cut_list,
+                   const std::vector<std::string>&column_list );
```

**Figure 2—1 Example of code change that impacts design by adding the method writeToFile and a dependency relationship between classes PyFitsController and QtCut**

2.1 Mapping Code Change to Design Change

We are interested in code changes that have a clear affect on the UML class diagrams representing the static design model of a software system. Examples of changes that impact the design include such things as addition/removal of a class, changes to the relationships between classes, and addition/removal of certain types of methods to a class. Specifically, we define design change as the addition or deletion of a class, a method, or a relationship (i.e., generalization, association, dependency) in the class diagram. These types of changes impact the structure of the diagram in a clear and
meaningful way with respect to the abstract design. Other types of changes are only related to implementation details and do not impact the class diagram in any meaningful way in the context of the design.

2.2 Changes that Impact Design

Changes to source code that involve addition or removal of a class, method, or class relationship can impact the class diagram. Figure 2—1 gives an example of code changes that affect the design. The figure is a snapshot from the diff output of two releases of the header file PyFitsController.h. In the newer release (1.19.1), one method, named `writeToFile`, has been added. As a result, this code change causes a corresponding design change, i.e., the addition of the new method `writeToFile` to class `PyFitsController`.

Additionally, in Figure 2—1, there is another user-defined type, `DataSource`, used in the parameter list of the new method. This produces yet another potential new dependency relationship between the classes `PyFitsController` and `DataSource`. But by analyzing the complete source code of `PyFitsController.h` release 1.18.1, we find that this dependency relationship already exists between these two classes. As a result, this specific code change does not affect the design; it just strengthens the dependency relationship.

Figure 2—2 is an example of a code change that results in a new association relationship between two classes. The code change is a part of the diff for the header file FitsNTuple.h releases 1.18.1 and 1.19.1. The code change shows the declaration of a new vector that uses the class `DataColumn`. To determine if this code change corresponds to addition of a new association between classes `FitsNTuple` and `DataColumn`, two
conditions are required. The first condition is the absence of this relationship in the older release. The second condition is the scope of the new variable. The declared variable must be a data member, i.e., this declaration must be in class scope. The analysis of the code change in Figure 2 shows that the new vector is declared as a data member in release 1.19.1. The analysis also shows that there is no data member of class DataColumn in release 1.18.1. Based on these two observations we can conclude that a new association relationship between FitsNTuple and DataColumn has been added to the design of HippoDraw release 1.19.1.

--- ../HippoDraw-1.18.1/fits/FitsNTuple.h
+++ ../HippoDraw-1.19.1/fits/FitsNTuple.h

Namespace hippodraw {
-  class FitsFile;
+  class DataColumn;
+  class FitsFile;
private:
-  std::vector<std::vector< double > * >m_data;
+  std::vector<DataColumn * >m_columns;

Figure 2—2. A code change that impacts the design due to the addition of a new association relationship between classes FitsNTuple and DataColumn.

2.3 Changes that do Not Impact Design

Many code changes pertain to implementation details and do not impact the design. In fact, most code changes should not impact the design; rather those changes should realize the design. This is particularly true during initial development or fault (bug) fixing. To correct a bug (i.e., not a design fault) source code is modified. This code change implements the design correctly and as such does not impact the class
diagram. Many bug fixes involve the modification of a loop or if-statement condition [Raghavan et al. 2004]. Changing a conditional impacts the implementation but not the design. Even some changes to class or method definitions do not necessarily lead to a design change. For instance, adding a new constructor function to a class does little to impact the design.

Figure 2—3 presents code changes that are generated by the diff utility of two revisions of the source file domparser.cpp, which is part of the KDE library. It is clear that these changes have no effect on the design of the software. No class has been added or deleted, the code changes do not show any addition or deletion of a method, and there is no code change that would affect the addition or deletion of any relationship. These changes do not require any updates to the corresponding UML class diagrams.

```diff
--- domparser.cpp (revision 731221)
+++ domparser.cpp (revision 731222)
@@ -82,7 +82,7 @@
case DOMParser::ParseFromString:
    {
       if (args.size() != 2) {
-          return Undefined();
+          return jsUndefined();
       }
       QString str = args[zero]->toString(exec).qstring();
@@ -101,7 +101,7 @@
    }
    return Undefined();
+   return jsUndefined();
}
```

Figure 2—3 Code changes from two revisions of domparser.cpp (KDE) that do not impact the design
Figure 2—4 shows three different examples of code changes between releases 1.18.1 and 1.19.1 of HippoDraw. In the first example Figure 2—4 Part A, the function \textit{getRank} is no longer declared virtual. The parameter of the function \textit{fillFromTableColumn} has been renamed from \textit{vec} to \textit{v} in the second example (B). In the third example (C), the macro \texttt{MDL\_QTHIPPOPLOT\_API} has been added to the class declaration. Even though these are changes to classes and methods they do not impact the design of the software. These changes are implementation changes and do not impact the UML class diagram of the design.

\begin{verbatim}
- virtual unsigned int getRank()const;
+ unsigned int getRank() const;

(A)

- int fillFromTableColumn(std::vector<double>&vec,
+ int fillFromTableColumn(std::vector<double>&v,

(B)

- class PickTable: public PickTableBase
+ class MDL_QTHIPPOPLOT_API PickTable: public PickTableBase

(C)
\end{verbatim}

\textbf{Figure 2—4} Three examples of code changes to method signatures that do not impact the design

\section*{2.4 Automatically Identifying Design Changes}

We implemented a tool, srcTracer (Source Tracer), to realize our approach to automatically identify design changes from code changes. The tool discovers when a particular code change may have broken the traceability to the design and gives details about what changed in the design. From these results the design document can be updated manually or by some future tool.
Output identifying design changes to a file appears as:

- **NEW METHOD** FitsNTuple::replaceColumn
- **NEW GENERALIZATION FROM** RootNtuple TO DataSource
- **OLD ASSOCIATION FROM** RootNTuple TO DataSourceException
- **NEW DEPENDENCY FROM** FitsNTuple TO DataColumn

The process begins with a code change that results in two versions of the source code. First, the source code of the two versions is represented in srcML [Collard, Kagdi, Maletic 2003], an XML format that supports the (static) analysis required. Second, the code change(s) are represented with additional XML markup in srcDiff [Maletic, Collard 2004] that supports analysis on the differences. Lastly, the changes that impact the design are identified from the code changes via a number of XPath queries. We now briefly describe srcML and srcDiff for continuity and focus in detail on identification of the design changes.

The srcML format is used to represent the source code of the two versions of the file. srcML is an XML representation of source code where the source code text is marked with elements indicating the location of syntactic elements. The elements marked in srcML include class, function (method), type, etc. Once in the srcML format, queries can be performed on source code using XPath, the XML addressing language. For example, the XPath expression “/unit/function” finds all function definitions at the top-level of the document. Another example is the XPath query “//function[name='convert']” finds the function definition with a name of convert. Figure 2—5 shows an example of source code and its corresponding srcML representation. Note that how each element is marked with its syntactic information.
// swap two numbers
if( a> b )
{
  t=a;
  a=b;
  b=t;
}

(A) Source Code

(B) srcML Representation

Figure 2—5 Source code of swapping two numbers (A) and its srcML representation (B). Syntactic elements are marked as tags around the original (escaped) source code text.

While the diff utility can easily collect source code changes, the output produced is purely textual information. It is very difficult to automatically recover the syntactic information of the code changes. To overcome this problem, srcDiff [Maletic, Collard 2004] is used. srcDiff is an intentional format for representing differences in XML. That is, it contains both versions of the source code and their differences along with the syntactic information from srcML. The srcDiff format is a direct extension of srcML. The srcML of two versions of a file (i.e., old and new) are stored.
<diff:common>
  <diff:old>
    <cpp:include># include &lt;../trial1&gt;</cpp:include>
  </diff:old>
  <diff:new>
    <cpp:include># include &lt;trial1&gt;</cpp:include>
  </diff:new>
</diff:common>

<comment type="block">/*a function*/
  <diff:old>2003</diff:old>
  <diff:new>2004</diff:new>
*/

<function>int sampleFunction(int a, int b, int c)
  <block>
    ...
    <diff:new>
      total = total + a;
      product = product * a;
    </diff:new>
    ...
  </block>
</function>

Figure 2—6. A srcDiff program fragment with selected srcML markup. The srcDiff tags are shown in bold.

The difference elements diff:common, diff:old, and diff:new represent sections that are common to both versions, deleted from the old version, and added to the new version respectively. Once in this format, the source code and differences can be queried using XPath with a combination of the difference elements (diff:*) and the srcML elements. Figure 2—6 presents a fragment of a srcDiff document. As shown in the figure one includes path of one file has been changes. Also new statements have been added to the body of the function sampleFunction. Examples of the srcDiff format are presented in the following sections as the change identification process is detailed.
2.4.1 Design Change Identification

Since the approach supports traceability from source code to design, the design change identification process depends on the syntactic information of the code change. This information can be extracted from the srcDiff representation of the code change. Once the code change has been identified as a design change, this design change is reported to keep it consistent with the code.

Design changes are identified in a series of steps, first added/removed classes, next added/removed methods, and lastly changes in relationships (added/removed generalizations, associations, and dependencies respectively). The information about a design change from a previous step is used to help identify the design change of the next step. For example, the code change in Figure 2—1 shows two types of design change, the addition of method `writeToFile` and the addition of dependency between classes `PyFitsController` and `QtCut`. The new method is identified first and is reported. Then, in the next step, the parameters of this new method are used to determine a new dependency relationship.

The process of identifying changes in code-design traceability is summarized in the following procedure:

1. Generate the srcML for each of the two file versions
2. Generate the srcDiff from the two srcML files
3. Query srcDiff to identify design changes
   a. Added/Deleted classes
   b. Added/Deleted methods
c. Added/Deleted relationships

4. Report the design change

We now discuss each of the identification steps in detail in the order that they occur. Also some detail on the XPath queries used to find the appropriate changes is provided. More examples and details about querying srcDiff are discussed in [Maletic, Collard 2004].

```
<diff:common>
<class>class <name>ColorBoxPointRep</name>
<block>

<diff:new>
<function_decl><type>virtual void</type>
    <name>setBoxEdge</name>({ bool show 
    });</function_decl>
</diff:new>

</block></class>
</diff:common>
```

Figure 2—7 The partial srcDiff of ColorBoxPointRep.h. A new method setBoxEdge was added

2.4.2 Classes and Methods

To identify if a code change contains an added/deleted class or method, the srcDiff of the differences is queried to find all methods and classes that are included in added or deleted code. In srcDiff, these are the elements that are contained in the difference elements diff:old or diff:new. Figure 2—7 shows a partial srcDiff of the file ColorBoxPointsRep.h. For clarity, only pertinent srcML elements are shown. The class ColorBoxPointRep exists in both versions, as indicated by being directly inside the
difference element `diff:common`. This class has a new method, `setBoxEdge`, indicated by being directly inside a difference element `diff:new`.

The general form of the XPath query to find new methods added to existing classes is:

```
//class[diff:iscommon()]/function_decl[diff:isadded()]/name
```

This query first finds all class definitions anywhere in the source code file. The predicate `[diff:iscommon()]` checks that the discovered classes exist in both versions of the document. The srcDiff XPath extension function `diff:iscommon()` is used here for clarity. Then within these existing classes it looks for method declarations (`function_decl`) that are new (checked with the predicate `[diff:isadded()]`). The final result of this query is the name of all methods added to existing classes. To find the names of all deleted methods, a similar XPath query is used, except instead of using the predicate `[diff:isadded()]` to find the added methods, we use the predicate `[diff:isdeleted()]` to find the deleted methods.

```xml
<diff:new>
  <class><name>VariableMesh</name>
  <block>
    <public>public:
      <constructor_decl> <name>VariableMesh</name>();</constructor_decl>
      <function_decl> <type><name>bool</name></type>
        <name>acceptFunction</name>();</function_decl>
    </public>
  </block>
</diff:new>
```

**Figure 2—8** The partial srcDiff of `VariableMesh.h`. The new class `VariableMesh` has been added.
The resulting queries find the names of these added/deleted methods, not the complete method signature, i.e., parameter number and types. We do not consider function overloading a design change. We are mainly concerned about the unique names of the methods. The new method is reported as a design change if the same name of that method does not exist in the old version of the source code. This means the name of the new function is unique.

To find new classes that are added to the source code (design), the following XPath query is used:

\[
//\text{class}[\text{diff:isadded()}/\text{name}
\]

The query finds the names of all classes that are tagged with the srcDiff element \textit{diff:new}. For example, Figure 2—8 shows partial of the srcDiff representation of the file \textit{VariableMesh.h}. The class \textit{VariableMesh} is added to the code via the change. The whole XML sub-tree of this added class (\texttt{<class> ...</class>}) is directly enclosed (or tagged) by the srcDiff tag \texttt{<diff:new>}. The same is applied for identifying deleted classes. The deleted classes are tagged with \texttt{<diff:old>} and they are identified by the predicate \texttt{[diff:isdeleted()]}.

### 2.4.3 Relationships

To identify changes in relationships, we designed queries to locate any change in the usage of non-primitive types (i.e., classes). For example, a declaration using class A is added to class B. This indicates a potential new relationship between the classes A and B. Alternatively, this may indicate a change to an existing relationship between the classes. The impact on the relationship of the usage of this type depends on where the
type change occurs. If the type change is in a super type then this indicates a change of a generalization relationship. If the type change is in a declaration within the scope of a method then this code change is identified as a new dependency relationship. And finally, if it is the declaration of a new data member (class scope) then it is an association relationship. The process of identifying changes in relationships is summarized as follows:

1. Query srcDiff to locate any added/deleted type (class) in the code.
2. If the added/deleted type has been used as a super type, then this is added/deleted generalization
3. If the added/deleted type has been used to declare a data member (class scope) then this is added/deleted association
4. If the added/deleted type has been used to declare a local member (method scope) then this is added/deleted dependency.

To identify added/removed generalizations, srcDiff is queried to check any change in the super types of the existing classes. Figure 2—9 shows how this change appears in srcDiff. The figure is the partial srcDiff from the file RootController.h. A new supertype, Observer, has been added to the existing class RootController.

The XPath query that is used to identify all new generalizations is:

```xml
//super//name[diff:isadded()]
```

This query finds the names of all added super types (classes) to the existing class. If there is a new generalization relationship between classes A and B, the XPath query is applied to the srcDiff representation of class A to identify B as an added super type.
Figure 2—9 The partial srcDiff of RootController.h. The supertype Observer forms a new generalization.

Figure 2—10 shows a potential new dependency. The figure is the partial srcDiff from the file DataView.cxx. The method `prepareMarginRect` of class `DataView` contains a new declaration for the variable `plotter`. The class `PlotterBase` is used in the type. The general form of the XPath query to identify the added dependencies is:

```
//function//type//name[diff:isadded()]
```

Figure 2—10 Partial srcDiff of DataView.cxx. The method has a new declaration that uses `PlotterBase`, producing a potential new dependency.
This query first finds all types used in methods, including the return type of the method. Then the names used in these types are found. The XPath predicate [diff:isadded()], ensures that these names were added. The resulting names are the destination (depends-on) of the dependency relationships.

The XPath query to identify potential added associations is:

```
//type//name[diff:isadded()]/src:isdatamember()
```

This query first finds all names used in a type that has been deleted, [diff:isdeleted()]. Then it checks to make sure that is in a class, i.e., that this declaration is a data member. The srcML XPath extension function [src:isdatamember()] checks the context of the type to make sure that it is in class scope.

Figure 2—11 shows a potential new association. The data member `m_columns`, which is a vector of type `DataColumn`, has been added to the class `FitsNTuples` in the new release of the code. This results in a potential association relationship between classes `FitsNTuple` and `DataColumn`.

```
<class>class <name>FitsNTuple</name> <block>{
  public:

    <diff:new>
    <type>std::vector&lt;<name>DataColumn</name>>*</type>
    <name>m_columns</name>;
    </diff:new>

}</block>;
</class>
</diff:common>
```

**Figure 2—11** Partial srcDiff of FitsNTuple.h. The class exists in both versions. The new data member uses type DataColumn forming a potential new association
The potential design change may not necessarily break the code-to-design traceability links. There could be more than one method in class A that uses local objects of type B. In this case the dependency relationship between A and B already exists. While this potential design change does not impact the dependency relationships, it does increase the strength of the dependency relationship between classes A and B.

```
+ PlotterBase* plotter = getPlotter();
+ float marginYTop = 30.0;
  if (m_plotter -> hasAxis (Axes::Z))
  {
    marginYTop = 70.0;
  }
+ const FontBase* titlefont = plotter->titleFont();
+ if (titlefont) {
+   marginYTop = marginYTop+titlefont->pointSize()-9.0;}
+ const FontBase* zfont = plotter->labelFont(Axes::Z);
+ float marginYBottom = 34.0;
+ const FontBase* labelfont = plotter->labelFont(Axes::X);
+ if (labelfont) {
+   marginYBottom=marginYBottom+labelfont->pointSize()+1.0;}
```

Figure 2—12 Partial diff output of code changes in the body of method DataView::prepareMarginRect. Two potential new dependencies have been added from class DataView to PlotterBase and to FontBase classes.

For example Figure 2—12 shows part of the code changes to the file graphics/DataView.cxx. These changes were applied to the body of the method DataView::prepareMarginRect. The change declared three FontBase objects (pointers) and one PlotterBase object (pointer). All these code changes will be identified as three potential new dependency relationships between DataView and FontBase and one potential dependency relationship between DataView and PlotterBase. The dependency between DataView and PlotterBase classes will not be reported (or recognized) as design
change because this relationship already exists. But the dependency between DataView and FontBase is reported as design change because it does not exist before. Moreover, the three occurrences are ignored and the design change is reported as new dependency between DataView and FontBase classes.

The check for uniqueness of the dependency and the association relationship is accomplished by further querying of srcDiff. For example, suppose that an added dependency on class B was found in class A. This added dependency is a potential design change, but we first need to determine if this dependency is new or not. To check if this relationship does not exist in the older version of the code, the following query is used:

```plaintext
//function //type//name[not(diff:isadded())][. = 'B']
```

The query looks at methods to find all names used in types that are part of the old document, \([not(diff:isadded)]\), and which are using class B. If this query returns with any results, then we know that the potential design change increases the number of occurrences of this dependency, but does not lead to a design change. If the result of this query is empty (i.e., no usage of class B was found) then it is a design change and requires an update of the design document.

### 2.5 Evaluation

To validate the approach we compare the results obtained automatically by our tool to the results of manual inspection by human experts. That is, the same problems are given to both the tool and the human experts. The objective is to see if the results obtained by tool are as good (or better) than the results obtained by the experts. Ideally,
one should not be able to discern the tool from the expert by a blind examination of the results.

We ran our tool over two complete releases of HippoDraw for this study. A subset of the changes was chosen and a set of problems was constructed so they could be presented to human experts. The details of the study and results are now presented.

2.5.1 Design Changes in HippoDraw

We used srcTracer to analyze code changes between releases 1.18.1 and 1.19.1 of HippoDraw. HippoDraw is an open source, object-oriented, system written in C++ used to build data-analysis applications. The srcTracer tool used libxml2 to execute the XPath queries. The tool took under a minute to run for the analysis, including the generation of the srcDiff format.

There are a total of 586 source and header files in release 1.18.1 of HippoDraw. When the system evolved from release 1.18.1 to release 1.19.1, there were 5389 new lines added and 1417 lines deleted (according to the unified diff format). These lines are distributed over 175 files of which 160 files have changes. The remaining 15 files include 13 files added to release 1.19.1 and 2 files deleted from release 1.18.1.

The identified design changes, from our tool’s results, are given in Table 2-1. Design changes are categorized according to the type of the change (class, method, relationship) and the context of the code change (in a new, deleted, or changed file). The plus symbol means added and the minus symbols means deleted. New files did not exist in release 1.18.1, while deleted files did not exist in release 1.19.1. There were 118 added methods to release 1.19.1. These 118 new methods were distributed as: 44
methods were added to new files and the remaining 74 methods were added to existing files. On the other hand, there were 9 methods deleted from release 1.19.1 and 8 of these methods were deleted from existing files.

Table 2-1  Design changes automatically identified in HippoDraw release 1.19.1 compared to release 1.18.1 by the tool srcTracer

<table>
<thead>
<tr>
<th>Design Change</th>
<th>New Files</th>
<th>Deleted Files</th>
<th>Changed Files</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>+</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Methods</td>
<td>+</td>
<td>44</td>
<td>0</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Generalizations</td>
<td>+</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dependencies</td>
<td>+</td>
<td>19</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Associations</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Design Changes</td>
<td>73</td>
<td>4</td>
<td>127</td>
<td>204</td>
</tr>
</tbody>
</table>

2.5.2 The Study

The study compares the results of the tool to that of manual inspection by human experts. Not only will this allow us to determine the accuracy of the tool, we also are able to determine the amount of time spent by experts. This gives us a relative feel for the time savings of using such a tool.

We were able to secure three developers who have expertise in both C++ and UML to act as subjects for the study. All subjects are graduate students and all have experience working in industry. All three are also very familiar with the design of the
studied system. For the purposes of this type of evaluative comparison, a minimum of two human experts is required.

Of the 175 files changes we selected a subset based on the following factors:

1. Variation of the changes - we attempted to cover most types of codes changes
2. Type of the change – we did not include files that contained non-code changes (e.g. comment changes)
3. File types - we selected situations where only one of header or the source file were changed

Given these criteria we selected 24 files from the 175 files for our study (randomly selecting as random as possible). For each of the 24 sets of changes (one set per file) we developed a problem to pose to the experts. We selected 24 file because it was over 10% of the changed files and 24 problems seemed a reasonable task to give to the experts. Our estimate of the time required to read and answer this number of problems manually was two hours.

For each of the 24 files, the standard unified diff format of the two versions was generated. Beside the code differences for each file, the design of the older release (1.18.1) for the code was provided as UML class diagrams of the source code under investigation. The design model for each file was reconstructed manually. A small description was given to each subject about the study and how they should go about answering the questions. The preparation of the study questions (reconstruction and drawing class diagrams) took approximately 40 hours. The Complete study is listed in APPENDIX B. Some minor changes have been made to the original diff output to keep
the size of the generated diff file reasonable and organized. For example, some unchanged code lines and comments have been deleted from the diff output.

--- ../HippoDraw-1.18.1/plotters/PlotterBase.cxx
+++ ../HippoDraw-1.19.1/plotters/PlotterBase.cxx

+ void PlotterBase::setBoxEdge ( bool flag ) { }
+ bool PlotterBase::getBoxEdge ( ) { return false; }

Bool PlotterBase::getShowGrid ( ) { return -1; }

+const FontBase *PlotterBase::titleFont ( ) const { return NULL; }
+FontBase*PlotterBase::labelFont ( hippodraw::Axes::Type axes ) const +{
  return NULL;
}

+bool PlotterBase::isImageConvertable ( ) const { return false; }
+bool PlotterBase::isTextPlotter ( ) const { return false; }

Does this code change add/delete.....in this UML class diagram?

- Classes: YES NO
- Relationships: YES NO
- Methods: YES NO

IF YES what are the changes?

Figure 2—13 One of the 24 test problems given to human experts. The line differences are the relevant output of the diff utility. The UML class diagram is the pertinent parts of the system before the design change. Based on this information, the expert answered the questions at the bottom regarding changes to the design.
Figure 2—13 shows one of the problems used the study. In this problem the code
differences of the two versions of the source file PlotterBase.cxx are given in the standard
unified diff format. The first version belongs to release 1.18.1 while the second version
belongs to release 1.19.1. The file PlotterBase.cxx is the source file (implementation) of
the class PlotterBase that is declared in the header file PlotterBase.h. The design of the
related parts of HippoDraw release 1.18.1 is represented as a UML diagram shown in the
figure. We ask two questions for each problem (given at the bottom of Figure 2—13).
We first ask if the code changes impact the given UML diagram. The second question
asks the user to write down any changes they perceive.

By showing the code changes and the design model of the source code together,
we directly examine the traceability between code evolution and design. For the example
in Figure 2—13 we can see that these code changes do impact the design and the
corresponding design document should be updated. Six new methods were added to class
PlotterBase that are not part of the given UML class diagram. A new dependency
relationship between class PlotterBase and class FontBase is also added that did not exist
in the original design model as can be seen in the UML diagram.

A brief description about the experiment was also given to the experts before they
start to solve the 24 problems. The description mainly defines the relationships between
classes. A key for the graphical notations (arrows) that are used to represent the
relationships was also given. This is necessary to unify their definitions for the different
types of relationships.

The definitions of relationships, as given to the experts, in the description are:
- Association from class A to class B: if A has a data member of type B.
- Aggregation between class A and class B: if A has a container data member that contains objects of B
- Dependency from class A to class B, if A has a method contains a local variable of type B

2.5.3 Results

The results obtained by the tool and the three experts for the 24 problems are given in Table 2-2. Each row represents the type of design change (e.g., class removed or added). The numbers in the table represents how many changes have been identified for each category.

The column Tool shows the results of the tool. Columns S1, S2, and S3 represent the results obtained by the three subjects. For example, the tool identified 33 added methods, while each of the three human experts (S1, S2, and S3) identified 32 added methods. To compare the human experts with the tool, the intersections of the results of each expert with the results of the tool are shown. For example, there are 33 added methods identified by the tool. The first expert (S1) identified 32 new methods. The intersection of these two sets shows that the tool also identifies the same 32 methods identified by that subject. The intersection column shows how closely the result of a subject is with that of the tool.

When the intersection is less than what the tool found, there are two possibilities. Either the tool misidentified a change as impacting the design or the expert overlooked a design change. To verify these two possibilities, we need to check the results of the other
experts. The second expert also identified 32 methods of the 33 (as the intersection shows). Did both subjects ignore the same method? If the answer is yes this means the tool may have misidentified a change. On the other hand if the answer is no, this means each expert overlooked a different method. The same thing can be said about the 32 methods identified by the third expert.

Table 2-2 The comparisons of the results between tool and the three human subjects

<table>
<thead>
<tr>
<th></th>
<th>Tool</th>
<th>$S_1$</th>
<th>$S_1 \cap$ Tool</th>
<th>$S_2$</th>
<th>$S_2 \cap$ Tool</th>
<th>$S_3$</th>
<th>$S_3 \cap$ Tool</th>
<th>$(S_1 \cap S_2 \cap S_3) \cap$ Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classes</strong></td>
<td>+</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>+</td>
<td>33</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td><strong>Generalizations</strong></td>
<td>+</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>+</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Associations</strong></td>
<td>+</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The rightmost column in the table gives the union of the subjects intersected with the tool. As can be seen, the subjects each missed an added method but not the same one. In this particular case, each of the three subjects missed a different method addition. Overlooking a design change in the code is not surprising as some changes are large and not easily followed. Tool support for this task will improve the quality of traceability and identifying design changes.

By comparing the experts’ results with the tool we found that the cumulative results for the experts were identical with the results from our tool. All design changes
individually identified by the subjects were a subset of the design changes identified by the tool. As such the tool performed better than each individual human expert and performed as good as the three subjects together.

With regards to effort spent, the three subjects required 80 minutes on average to complete the 24 problems. It should also be considered that the problems were presented in a very clear and straightforward manner with all the associated information (UML and code). This is a best-case scenario for manually evaluating changes and in practice there would be a large amount of time spent putting all this information together to assess the change. Our tool took less than one minute to run against the entire system.

2.6 Threats to Validity and Limitations

The evaluation of the approach as to the accuracy of the tool (i.e. user study) covers only one system. However, it is unclear if different systems would impact the results greatly. It is possible software addressing different domains would display different evolutionary trends and more complex changes. In this case results could be affected and further studies are warranted. But HippoDraw is in a fairly general domain and the types of changes incremental. We see no serious reason that the accuracy of our results presented here will not scale to other like systems and changes. Our selection of the files was not done completely randomly however we did assure for a diversity of problems. We could have an expert check every change but there is no indication that the results would differ.

Although the evaluation study was between two releases of the system, the granularity of the changes was not very large. It still remains to validate the approach on
large code variations. However, these changes may be very difficult for subjects to comprehend. The amount of time and information for subjects to comprehend and accurately assess the changes are most likely to increase.

The approach was implemented using the srcML and srcDiff translators. The srcML translator is based on an unprocessed view of the software (i.e., before preprocessor is run), and does not take into account expansion of preprocessing directives. However, this was not an issue for HippoDraw as few changes involved complicated preprocessor directives or macros. If the software system under review did incur many changes to preprocessor directives and macros, the tool can be applied to both the unprocessed and preprocessed code.

The approach was only applied to C++ and not tested on other object-oriented programming languages. However, the srcML and srcDiff formats do support Java and we expect our work will map to other languages

2.7 Related Work

The related work focus into three main groups; software evolution (mainly source code), design changes and evolution, and code-design traceability.

2.7.1 Software Evolution

Leman was among the first researchers who studied the evolution of software systems since the early 70s. During his long study, he developed a set of laws known as the laws of software evolution. These are widely known and considered as standards for evolution. These laws are recently reformulated and published in [Lehman, Ramil 2001]
and [Lehman, Ramil 2004]. The laws highlight the facts that software systems continuously change and grow. During this evolutionary growth, the complexity is increased and the quality is declined.

Godfrey and Tu [Godfrey, Tu 2000] analyzed the growth in the size (LOC) of the Linux kernel. They found that Linux has been growing at a super-linear rate for several years. A similar study on the GNOME project is discussed by Koch and Schneider [Koch, Schneider 2002]. They found the project has steadily increase in size. Nakakoji et al. [Nakakoji et al. 2002] studied the evolutionary patterns and communities in four open source systems. They proposed to classify OSS into three types: Exploration-Oriented, Utility-Oriented, and Service-Oriented.

Paulson et al. [Paulson, Succi, Eberlein 2004] compared the growth of open-source and closed-source software projects. They found that the growth of both types is similar. Herraiz et al. [Herraiz, Robles, Gonzalez-Barahona 2006] used SLOC and file counts to compare the evolution of 12 open source projects. They found the growth pattern of these two measures is the same. Robles et al. [Robles, Gonzalez-Barahona, Merelo 2006] studied the evolution of LOC and file types in KDE. They found that the growth pattern is super linear for all file types under study. Kothari et al. [Kothari, Shokoufandeh, Mancoridis, Hassan 2006] studied the evolution of code stored in software repositories. They identified change clusters to help in classifying code changes activities.

My research does not support code evolution only. It supports the evolution of code and design together. Most of the related work in this area investigates any
evolutionary patterns in code. My research investigates evolutionary patterns in code and design together.

2.7.2 Design Evolution

Kim et al. [Kim, Notkin, Grossman 2007] presented an automated approach to infer high level structural changes in software. They represent the structural changes as a set of change rules. The approach infers the high level changes from the changes in method headers across program versions. Weißgerber and Diehl [Weiβgerber, Diehl 2006] presented a technique to identify refactorings. Their identification process is also based on comparing method signatures and full name of fields and classes. Both of these approaches do not support changes to relationships. A comprehensive survey about refactoring techniques is presented in [Mens, Tourwe 2004].

Xing and Stroulia [Xing, Stroulia 2005b; 2007b] presented an algorithm (UMLDiff) which automatically detected structural changes between the designs of subsequent versions of OO software. The algorithm basically compares the two directed graphs that represent the design model. Design models are constructed from the abstract syntax tree (AST) of the source code. UMLDiff was used in [Xing, Stroulia 2004b] to study class evolution in OO programs. They provided taxonomy of class-evolution types based on their changed history. In [Xing, Stroulia 2004c] and [Xing, Stroulia 2005a] UMLDiff was used to analyze the design-level structural changes between two subsequent software versions to understand the phases and the styles of the evolution of OO systems. They classified system evolution based on the structural modifications. In [Xing, Stroulia 2007a] Xing and Stroulia also studied and supported the evolution of API.
Other examples of graph comparison approaches are presented in [Apiwattanapong, Orso, Harrold. 2004] and [Nguyen 2006].

Fluri and Gall [Fluri, Gall 2006] identified and classified source code changes based on tree edit operations on the AST. We do not compare two design models. These comparisons primarily use graph comparison, which is not efficient. Additional matching techniques are discussed in [Kim, Notkin 2006].

A recent survey about understanding software maintenance and evolution by analyzing individual changes is presented by [Benestad, Anda, Arisholm 2009]. The review summarizes change attributes that have been used in empirical investigations, and propose a conceptual model for change-based studies that enables us to classify the attributes.

My work is distinguished from related work in this area by the identification of design changes from small and incremental code changes. Most design differencing tools depends on graph comparison. The two design models, or design artifacts, are represented in some form of graphs, e.g., AST. However, graph comparison is not an easy task. Furthermore, many of these tools depend on statistical calculations and thresholds. By using the proposed approach the design differences can found without comparing the two designs or two ASTs. Instead the code changes are generated, the design changes are identified, and then changes can be applied to the original design model to get the new design. The approach works even if the initial design and source code are not consistent.
2.7.3 Recovering Traceability Links

The results of a recent industrial case study [Feilkas, Ratiu, Jurgens 2009] showed that the informal documentation and the source code are often not kept consistent with each other during evolution and none of them completely reflects the intended architecture. Developers also often are not completely aware of the intended architecture.

Antoniol et al. [Antoniol, Caprile, Potrich, Tonella 2000; Antoniol, Caprile, Potrich, Tonella 2001] presented an approach to trace OO design to implementation. The goal was to check the compliance of OO design with source code. Design elements are matched with the corresponding code. The matching process works on design artifacts expressed in the OMT (Object Modeling Technique) notation and accepts C++ source code. Their approach does support direct comparison between code and design. To be compared, both design and code are transformed into intermediate formats (AST).

Antoniol et al. [Antoniol, Di Penta, Merlo 2004] proposed an automatic approach, based on IR (Information Retrieval) techniques, to trace, identify and document evolution discontinuities at class level. The approached has been used to identify cases of possible refactorings. The work is limited to refactorings of classes (not methods or relationships).

An approach and a prototype called Zelta were proposed by [Ratanotayanon, Sim, Raycraft 2009]. The aim of the approach is to capture and manage traceability links of features to multiple artifacts (documents and source code). The approach is light weight and basically depends only on the textual contents of artifacts (not design contexts).
IR techniques are used in many approaches to recover traceability links between code and documentation. Antoniol et al. [Antoniol, Canfora, Casazza, De Lucia 2000; Antoniol et al. 2002] proposed methods based on IR techniques to recover traceability between source code and its corresponding free text documentation. In [Antoniol et al. 2000] they traced classes to functional requirements of java source code. An advanced IR technique using Latent Semantic Indexing (LSI) has been developed by Marcus and Maletic [Marcus, Maletic 2003] to automatically identify traceability links from system documentation to program source code. De Lucia et al. [De Lucia, Fasano, Oliveto, Tortora 2005; 2007; De Lucia, Oliveto, Tortora 2008] used LSI techniques to develop a traceability recovery tool for software artifacts. Zhou and Yu [Zhou, Yu 2007] considered the traceability relationship between software requirement and OO design. Zhao et al. [Zhao et al. 2003] used IR combined with static analysis of source code structures to find the implementation of each requirement.

All IR techniques are statistical and the correctness of the results depends on the performance of the matching algorithm. They also require considerable effort to retrieve information from code and documents. These methods do not provide traceability between code and UML design specifications. Hayes et al. [Hayes, Dekhtyar, Sundaram 2006] studied and evaluated different requirements tracing methods for verification and validation purposes.

Reiss [Reiss 2002; 2005] built a prototype supported by a tool called CLIME to ensure the consistency of the different artifacts, including UML diagrams and source code, of software development. Information about the artifacts is extracted and stored in
a relational database. The tool builds a complete set of constraint rules for the software system. Then, the validity of these constraints is verified by mapping to a database query. The approach does require a predefined set of design constraints while our approach does not. Our approach also supports incremental small code changes. We directly analyze the specific code change instead of the changes at the file level (Riess’s approach). A more specific rule based approach to support traceability and completeness checking of design models and code specification of agent-oriented systems is presented in [Cysneiros, Zisman 2008].

In this area my work is distinguished from this related work in multiple ways. The techniques used are lightweight and not IR or rule based. Only the code changes are analyzed. There is no comparison between the code and a design document/artifact. Unlike most of the others, the changes in the relationships between classes in the design are discovered from just code.

2.8 Summary

An approach has been developed that automatically determines if a given source code change impacts the design (i.e., UML class diagram) of the system. This allows code-to-design traceability to be consistently maintained as the source code evolves. The approach uses lightweight analysis and syntactic differencing of the source code changes to determine if the change alters the class diagram in the context of abstract design. The intent is to support both the simultaneous updating of design documents with code changes and to bring old design documents up to date with current code given the change history.
An efficient tool was developed to support the approach and is applied to an open source system (i.e., HippoDraw). The results are evaluated and compared against manual inspection by human experts. The tool performs better than (error prone) manual inspection in a fraction of the time.
CHAPTER 3

Changes in Code-to-Design Consistency during Evolution

The automatic identification of breaks in code-to-design traceability from specific code change has been presented and discussed in CHAPTER 2. From single changes to design for a single version, we are now examining how multiple changes (commits) impact the design of a system over a duration of the version history (i.e. breaking code-design traceability).

The approach and the supporting tool srcTracer (discussed in CHAPTER 2) have been used to conduct a detailed empirical investigation to understand how daily and incremental changes to source code (i.e., commits) impact the design of a software system. The goal is to investigate how often the code-design consistency is broken during the evolution of the system. The main benefit from this understanding is the identification of design stability (and instability) periods.

We need to systematically understand what percentage of commits actually impacts the design. The tool srcTracer is used to determine what syntactic elements changed due to a commit. From this analysis, the commits are categorized as impacting the design or not. Figure 3—1 shows an example of code change that impacts design by adding/deleting three generalizations. This is part of the code changes committed on the file kateprinter.h in revision 723866 on Kate. So, this revision (commit) is considered as a design impact commit.
Figure 3—1 Part of code changes in revision 723866 on Kate. The change impact impacts design of Kate by adding/deleting three generalizations.

We are interested in investigating how often the design is impacted or changed. Whereas changes that simply modify a condition typically do not impact the design of a system and are more likely to be bug fixes. The percentage of commits that impact design and their distribution over time are presented and discussed.

The chapter has been addressed and accepted for publication in an invited submission to the Software Quality Journal [Hammad, Collard, Maletic 2010a]. In the rest of this Chapter, information about the collected data for the empirical study is presented in Section 3.1. Categorization of commits based on design impact is presented in Section 3.2. Section 3.3 discuses design stability based on history of design changes. Characteristics of commits are presented in Section 3.4. Section 3.5 investigates the relationship between design changes and bug fixes. Commit labeling technique is presented in Section 3.6. Related work is discussed in Section 3.8.
3.1 Data Collection

The commits of four C++ open source projects over specific time durations are extracted and analyzed. The four projects are; the KDE editor Kate (http://kate-editor.org), the KOffice spreadsheet KSpread (www.koffice.org/kspread), the quantitative finance library QuantLib (www.quantlib.org), and the cross-platform GUI library wxWidgets (www.wxwidgets.org). These projects have been chosen because they are C++ (object oriented), well documented, have large evolutionary histories, and vary in their purposes.

The basic unit of data under investigation is the commit and its related information. Commit is an operation done by developers to save changes they have made. Most open source projects use version control systems to manage changes done by developers. So, the historical changes (versions) of files are saved for future use. The two widely used version control systems are Concurrent Versions System (CVS) and Subversion (SVN). SVN is recent and many CVS controlled repositories are transformed to SVN. The repositories of four projects under the study are managed by SVN.

```
r502508| rodda |2006-01-26 05:49:03 -0500 (Thu,26 Jan 2006) | 3 lines
Changed paths:
   M /trunk/KDE/kdelibs/kate/TODO
   M /trunk/KDE/kdelibs/kate/part/kateviewhelpers.cpp
Fix drawing of word wrap indicators (although we should probably get some nicer graphics instead)
```

**Figure 3—2 The log message of revision 502508 on Kate extracted from the Subversion repository**
For the purposes of the study, we need (for each revision) to extract changed files, date when the revision was committed, and the commit message.

For example, Figure 3—2 shows the extracted log message (record) for revision 502508 on Kate. The log record in Figure 3—2 shows the revision number (r502508), the committer ID (rodda), the date of the commit (2006-01-26), the names of changed files with their full paths, and finally the commit message. Log message can also be extracted as XML format to ease the process of data collection and analysis. Figure 3—3 shows the XML representation of a log message in Figure 3—2. The action property (“M”) means that the file has been modified by this commit. The other two actions are added (“A”) which means the file had been added and deleted (“D”) which means the file has been deleted by the commit.

```xml
<log>
  <logentry revision="502508">
    <author>rodda</author>
    <date>2006-01-26T10:49:03.614968Z</date>
    <paths>
      <path action="M">/trunk/KDE/kdelibs/kate/part/kateviewhelpers.cpp</path>
      <path action="M">/trunk/KDE/kdelibs/kate/TODO</path>
    </paths>
    <msg>Fix drawing of word wrap indicators (although we should probably get some nicer graphics instead)</msg>
  </logentry>
</log>
```

Figure 3—3 The extracted XML representation of log message in Figure 3—2
For each project all commits between two specific dates were extracted to cover a period from the evolutionary history of the project. Commits were extracted from a specific part (i.e. target directory) in each project. We carefully selected these directories which contain the most source and header C++ files. For example, in the wxWidgets project we extracted the commits from the Subversion directory (http://svn.wxwidgets.org/svn/wx/wxWidgets/trunk). Start and end dates were selected to cover three consecutive years time durations. The starting dates were chosen so that all the projects were well-established and were undergoing active development and maintenance.

From the set of overall commits in that time duration, we selected a subset for analysis. Since the analysis was on design changes caused by C++ code changes, commits with no C++ code changes were excluded. Within each commit, we also excluded non C++ source or header files. Test files were not included since they are not part of the overall design.

Table 3-1 For each studied project, the directory the files came from, the time duration over a 3 year period, the total number of source files at the beginning of the time period, and the total studied commits (after filtering)

<table>
<thead>
<tr>
<th>Open Source Project</th>
<th>Directory</th>
<th>Time Period</th>
<th># Source Files</th>
<th>Total Commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>KDE/kdelibs/kate/</td>
<td>3 years (1/1/2006–12/31/2008)</td>
<td>111</td>
<td>1592</td>
</tr>
<tr>
<td>KSpread</td>
<td>koffice/kspread/</td>
<td>3 years (1/1/2006–12/31/2008)</td>
<td>207</td>
<td>2389</td>
</tr>
<tr>
<td>QuantLib</td>
<td>trunk/QuantLib/ql/</td>
<td>3 years (1/1/2006–12/31/2008)</td>
<td>671</td>
<td>2701</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>wxWidgets/trunk/</td>
<td>3 years (1/1/2005–12/31/2007)</td>
<td>3451</td>
<td>11438</td>
</tr>
</tbody>
</table>
Table 3-1 shows the total number of studied commits (after filtering), time duration, directory in the repository, and the number of C++ header and source files in that directory at the beginning of each time duration for the four projects. For example, for the KSpread project we extracted the commits from the directory koffice/kspread/ over a period of three years starting on the first day of 2006. At the first revision, there were 207 C++ header and source files. The total number of extracted commits that were included in the study for the KSpread project is 2389.

### 3.2 Categorization of Commits

The tool srcTracer and the approach presented in CHAPTER 2 were used to analyze the code changes of all the extracted commits (Table 3-1). The code change of each commit was analyzed individually (i.e. independently from other commits). Based on this analysis, the commits are categorized as either impacting the design or not. The percentages of commits with design impact for the four projects are presented in Table 3-2.

#### Table 3-2 Percentages of commits with design impact and with no design impact for the four projects during the three year time period

<table>
<thead>
<tr>
<th>Project</th>
<th>% Commits - Design Impact</th>
<th>% Commits – No Design Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>424 /1592 (27%)</td>
<td>1168 /1592 (73%)</td>
</tr>
<tr>
<td>KSpread</td>
<td>681 /2389 (29%)</td>
<td>1708 /2389 (71%)</td>
</tr>
<tr>
<td>QuantLib</td>
<td>924 /2701 (34%)</td>
<td>1777 /2701 (66%)</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>2269 /11438 (20%)</td>
<td>9169 /11438 (80%)</td>
</tr>
</tbody>
</table>
Figure 3—4  Histograms of commits to (a)Kate, (b)KSpread over 3 year time period. For each month commits are categorized based on impact to design. In almost all months the number of commits with design changes is much less than the number of commits with no impact to design.
Figure 3—5 Histograms of commits to (a) QuantLib and (b) wxWidgets over 3 year time period. For each month commits are categorized based on impact to design. In almost all months the number of commits with design changes is much less than the number of commits with no impact to design.
During the selected time window, most of the commits to these four projects did not impact the design. Among the 2701 extracted commits of QuantLib, there are 924 commits (34%) that changed the design in some way. On wxWidgets project, only 20% of the commits changed the design. For both Kate and KSpread the percentage is also low (27% and 29% respectively).

The distribution of commits over the 36 months for the four projects is given in Figure 3—4 and Figure 3—5. Each column in the chart represents the total number of commits during a specific month. The lower part of each column represents the number of commits that impacted the design during that month. The upper part of the column represents the number of commits that did not change any design element. For example, in February 2006 (Month 2) there were 63 commits to KSpread. Out of those commits the design of KSpread was changed in 12 of them. It is clear from Figure 3—4 and Figure 3—5 that in almost all the studied months, the number of design changes is less than other changes. More specifically, in all 144 months (36 months for each project) there were only 6 months in which design changes were more prevalent. This leads us to conclude that most of code changes do not impact the design.

3.3 Design Changes and Design Stability

From the discussed charts, we can identify periods of time where major design changes occurred. For example, most of the design changes (70%) to QuantLib occurred during the first half of the three year period. This may mean that the system was undergoing major development activities. Then in the second half of the three years the design was more stable with small maintenance activities occurring. There were 1798
new classes added during the first half of the three years comparing with 759 classes in the second half. Another example is the number of new features added to the system. By examining the history of the releases notes, we found that during the first half (January 2006 – June 2007), there was 6 releases with 70 new features compared to 4 releases with 51 new features for the second half. From the release notes we observed that the second half had more fixes (11 fixes) than the first half (2 fixes).

One application of categorizing commits based on design impact is for an indicator of the stability of code-to-design traceability (design stability) during maintenance/development activity. This indicates how much of this activity was focused on the design (i.e., potentially breaking traceability links). For example, in February 2006 (Figure 3—4-b) the percentage of commits that impacted the design of KSpread is 19% (12/63). The code-to-design traceability links were very stable during these commit activities. By using such measure, we can determine periods of design stability in the evolution history of the project.

The distribution of design impact commits with respect to the type of the impact is given in Table 3-3 and Table 3-4 for the four projects. Renamed classes were excluded using a simple check to see if the added and deleted classes in the commit have the same methods. This can also be identified directly from the srcDiff format by checking if the name of the class is the only syntax change.

The first column of each table is the type of design impact. The second two columns (+ and -) are the number of added or deleted elements for each design change.
The (# Commits) column is the total number of commits for each design type (or design element).

Table 3-3  Distribution of commits based on their design impact category on Kate and KSpread

<table>
<thead>
<tr>
<th>Design Impact</th>
<th>Kate</th>
<th>KSpread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Classes</td>
<td>94</td>
<td>70</td>
</tr>
<tr>
<td>Methods</td>
<td>165</td>
<td>727</td>
</tr>
<tr>
<td>Generalizations</td>
<td>89</td>
<td>81</td>
</tr>
<tr>
<td>Dependencies</td>
<td>311</td>
<td>266</td>
</tr>
<tr>
<td>Associations</td>
<td>143</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 3-4  Distribution of commits based on their design impact category on QuantLib and wxWidgets

<table>
<thead>
<tr>
<th>Design Impact</th>
<th>QuantLib</th>
<th>wxWidgets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Classes</td>
<td>255</td>
<td>422</td>
</tr>
<tr>
<td>Methods</td>
<td>719</td>
<td>135</td>
</tr>
<tr>
<td>Generalizations</td>
<td>166</td>
<td>671</td>
</tr>
<tr>
<td>Dependencies</td>
<td>365</td>
<td>766</td>
</tr>
<tr>
<td>Associations</td>
<td>277</td>
<td>769</td>
</tr>
</tbody>
</table>

For example, during the studied time window 525 commits impacted the design of KSpread by adding or deleting a method. These 525 commits added 2673 methods and deleted 2533 methods from the design of KSpread. Many commits have multiple design impacts. For example, the commit of revision 727209 in Kate caused changes to five files and had the following impacts on the design (as reported by srcTracer tool):
NEW GENERALIZATION FROM KateLayoutCache TO QObject
NEW DEPENDENCY FROM KateLayoutCache TO KateEditInfo
NEW METHOD KateLayoutCache::slotEditDone
OLD METHOD KateLayoutCache::slotTextInserted
OLD METHOD KateLayoutCache::slotTextRemoved
OLD METHOD KateLayoutCache::slotTextChanged
OLD METHOD KateEditHistory::editUndone

It is important to point out that the totals in Table 3-3 and Table 3-4 are based on the changes for each commit individually. For example, one design element could be added by one commit and removed by another. So, the total number of methods added to Kate is not 1654. This number indicates the total number of methods added by each commit independently from other commits. The distribution of commits, based on the design impact type, shows that API changes have the highest number of commits for the four projects.

More specifically, we can identify months (periods) where specific design changes occurred in large number. For example, the month which has the largest number of added classes during the last three years. This may indicate that the structure of the design is changing (major design change) and more new features are being added. On the other hand, periods of time with more API changes (methods) and very few added/deleted classes would mean that the functionality of the system is growing (or changing) but the features are stable. Other useful historical information is periods of time where the degree of coupling/cohesion is increased or decreased. The number of
added/deleted relationships could be used as indicator (or measure) for changes in the degree of coupling/cohesion.

For example in Kate, the month with largest number of added/deleted design elements is May-2007 (minimum design stability). During this month, 514 design elements have been added and deleted. The distribution of these 514 design changes over different design categories are shown in Table 3-5. The detailed numbers of added and deleted elements for each design category and possible interpretations that can be inferred are also shown.

The rank of the month shows the rank of May-2007 among all other 36 months based on the totals for each design category. For example, May-2007 had the largest total number (ranked 1) of added/deleted methods (367) among all studied months. On the other hand it is ranked third based on total number of added/deleted generalizations (12).

Table 3-5 Distribution of Design changes occurred in May 20007 on Kate. The month has the largest number of design changes. Possible interpretations can be identified from detailed changes.

<table>
<thead>
<tr>
<th>Design Impact</th>
<th>#Elements</th>
<th>Month Rank</th>
<th>Possible Interpretations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>-</td>
<td>Total</td>
</tr>
<tr>
<td>Classes</td>
<td>8</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>Methods</td>
<td>275</td>
<td>92</td>
<td>367</td>
</tr>
<tr>
<td>Generalizations</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Dependencies</td>
<td>54</td>
<td>32</td>
<td>86</td>
</tr>
<tr>
<td>Associations</td>
<td>10</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>351</td>
<td>16</td>
<td>514</td>
</tr>
</tbody>
</table>
Many interpretations can be inferred from analyzing design changes. During this month (May 2007) the design went over major changes. This month has the largest number of added/deleted classes which directly affect the structure of the design. Also, the same month has the largest number or changed methods. More methods have been added than deleted, which means growth in functionality. More generalizations have been deleted than added, which means big design architectural organization activities. The coupling between classes has increased because of the added dependencies (more than deleted). Finally, more associations have been deleted which would mean some classes became more independent (increase cohesion). Another interpretation is decrease in coupling. Collectively, more relationships (generalizations, dependencies and associations) have been added than deleted. So, in general the coupling has been increased by the end of May 2007.

3.4 Characteristics of Commits

After categorizing commits according to their impact on design, we are interested in studying the characteristics of each category. The two properties under investigation are the original size of the commit (number of changed files) and the number of changed (added and deleted) lines in the header and source files. The number of changed lines was calculated by using the unified format of the diff utility. The averages of these two properties for the four projects during the 3 years are shown in Table 3-6.
Table 3-6  For each project, the average size of the commit (number of files) and the average number of lines changed (i.e., added and deleted)

<table>
<thead>
<tr>
<th>Project</th>
<th>Commits</th>
<th>Impact on Design</th>
<th>No Impact on Design</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Avg. #Files</td>
<td>Avg. #Lines</td>
</tr>
<tr>
<td>Kate</td>
<td></td>
<td>13</td>
<td>217</td>
</tr>
<tr>
<td>KSpread</td>
<td></td>
<td>9</td>
<td>197</td>
</tr>
<tr>
<td>QuantLib</td>
<td></td>
<td>16</td>
<td>510</td>
</tr>
<tr>
<td>wxWidgets</td>
<td></td>
<td>7</td>
<td>1148</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td><strong>11</strong></td>
<td><strong>518</strong></td>
</tr>
</tbody>
</table>

For the four projects, the average of the number of changed lines by the design impact commits is significantly higher than the commits with no design impact. For example, in KSpread the average number of lines changed (added and deleted) is 197 per commit with design impact and 28 lines per commit with no impact. The average number of changed files per commit is also higher in commits with a design impact, except for KSpread where it is slightly lower (from 10 down to 9).

The distribution of commits, based on the number of changed files (size) is shown in Figure 3—6. Figure 3—7 shows the distribution of commits based on the number of changed (added/deleted) lines of code. The numbers of lines and files for all commits are used to create four quartiles for each project, with the value on the x-axis indicating the high end of the quartile. The distribution of commits across these quartiles is shown. In each quartile, the commits are categorized based on impact to design. The percentage of each category (design impact vs. no design impact) is calculated from the total number of commits for that category, i.e., all design commits from all four quartiles add up to 100%.

For example, in Kate 42% (Figure 3—6) of the commits have one to four lines of code
changed (first quartile). These commits are distributed as 2% with design impact and 40% with no impact.

Figure 3—6 The distribution of commits into quartiles (x-axis values) based on the number of changed files per commit. The percentages of commits with design impact and with no impact based are shown. In general, commits with a design impact have a larger number of changes files.
Figure 3—7 The distribution of commits into quartiles (x-axis values) based on the number of changes (added/deleted) lines of code per commit. The percentages of commits with design impact and with no impact based are shown. Commits with a design impact consist of more changed lines.

Also in Kate (Figure 3—7), only a single C++ source code file was changed in 60% of the commits with no design impact. On the other hand, only 6% of the design commits changed only a single file. Also in Kate, 63% of the design commits changed between 49 and 9787 lines of code. But only 11% of the commits with no impact are in
the same range. For files, an average of 65% of design impact commits occurred in the third and fourth quartiles (larger number of files) compared to 27% for the other commits. For lines, an average of 87% of design impact commits occurred in the third and fourth quartiles (larger number of lines) compared to 35% for commits with no impact.

In summary, commits with design impact contain a larger number of files and more lines of changed code.

Another characteristic is the amount of design impact caused by commits with design changes. By analyzing the number of added/deleted design elements for each commit, we can measure the impact of each commit on design. In other words, we want to measure the influence of commits on design. Commits with big influence are considered key commits in the evolutionary history of design. Key commits means that big or major design changes occurred. This change could be to add/delete a major feature or could be to reorganize the structure of the design.

Table 3-7 shows some statistics for the minimum and maximum number of added/deleted design elements for all commits with design impact in the four projects. For Kate, there are 140 commits have added or deleted only one design element (min impact). The percentage of these commits is 33% (140/424) of all commits with design impact (424). The maximum impact on design was done by one commit that added/deleted 244 design elements. The average added/deleted design elements for all the commits with design impact on Kate are 8.4. It is interesting that 81% of commits with design impact have added/deleted design elements less than the average (8.4).
other three projects follow the same pattern. Most design changes are small. Only few percentages of commits have extensive impact on design. In all four projects, only one commit has the maximum number of design impacts. The analysis of Table 3-7 shows that most commits with design impact have small impact on design.

<table>
<thead>
<tr>
<th></th>
<th>Min impact</th>
<th>Max impact</th>
<th>Average</th>
<th>% Commits with impact &lt; Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>1 (33%)</td>
<td>244</td>
<td>8.4</td>
<td>81%</td>
</tr>
<tr>
<td>KSpread</td>
<td>1 (24%)</td>
<td>692</td>
<td>12.5</td>
<td>80%</td>
</tr>
<tr>
<td>quantLib</td>
<td>1 (18%)</td>
<td>1745</td>
<td>23.6</td>
<td>82%</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>1 (35%)</td>
<td>652</td>
<td>10</td>
<td>84%</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>28%</strong></td>
<td><strong>-</strong></td>
<td><strong>13.6</strong></td>
<td><strong>82%</strong></td>
</tr>
</tbody>
</table>

Revision 602469 has largest number of added/deleted design elements on Kate. The commit message of this revision is: "Updated implementation of code completion to match interface changes. Still need to port the word completion plugin". This revision seems that it performed a big maintenance task to adapt with major change in the system (interface changes). This revision has updated 26 files, added 494 lines of code and deleted 1404 lines. Additionally, this revision impacted design by deleting 6 classes, adding 186 methods, deleting 29 methods, adding one generalization, deleting 6 generalizations, adding one dependency, deleting 5 dependencies and deleting 10 associations. The total of these design impacts is 244.

Another example is revision 9926 on QuantLib. It has the largest number of impacts. The commit message of this revision is: "Changed folder names to lowercase
in the source tree Changed folder names to lowercase in the source tree”. This means that the design went over a big refactoring task after major renaming occurred in the system. This revision has changed 1175 files, added 91677 new lines and deleted 62 lines.

In summary most design changes are incremental and small. Only some commits have big design impact (major breaks in code-design consistency). This indicates that design only undergoes extensive maintenance tasks occasionally.

3.5 Bug Fix vs. Design Change

The investigation of the analyzed commits leads to a key research question. If the code change does not impact the design, then what is it? Or more specifically, if a code change does not impact the design, then is it a bug fix? To address the later of these questions, we need to figure out which commits are bug fixes. In order to determine this we applied a simple analysis of the commit messages following the same technique used by [Mockus, Votta 2000] and [Pan, Kim, E. James Whitehead 2009]. Using this technique, certain key words in the commit message determine the type of the code change. The approach used in [Pan, Kim, E. James Whitehead 2009] checks if the log change contains “bug”, “fix”, or “patch” to determine if the revision is a bug fix. This technique is useful for Kate and KSpread due to the commit rules of KDE where any bug fix should be marked in the commit message by the key word “BUG” combined with the bug number. For example the commit message of revision 879107 on KSpread is “fix crash: invalid layout access. Thanks to Fbio Rodrigo for the patch! BUG: 171027”. This commit fixed the BUG 171027. Another big fix is revision 883749 on KSpread. The
commit message of this revision is "fix range handling * remove 'valid' property of katevirange * move validation of position/range to handleKeypress()". This message does not contain the keyword "bug", but we identify it as a bug fix from the keyword "fix".

In our study, the same extracted commits are grouped into two categories based on if the commit message contains the words "bug", "fix" or their derivations (e.g. fixed, fixing, bugs...etc). Then, we looked at how many commits in each group impacted the design. The result of this analysis is given in Table 3-8.

Among the 1592 commits extracted from Kate, there are 515 (32%) commits categorized as bug fixes. Based on our previous analysis 424 commits out of the 1592 are design changes. There are only 97 commits among these 424 (23%) commits that are categorized bug fixes. A quick look to the percentages of commits that are design change and bug fix (fourth column) leads to the conclusion that most of design changes are not bug fixes (less than 20% in average).

<table>
<thead>
<tr>
<th></th>
<th>#Bug Fix Commits</th>
<th>#Design Commits</th>
<th>Design ∩ Bug fix</th>
<th>Not Design ∩ Bug fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>515 /1592 (32%)</td>
<td>424 /1592 (27%)</td>
<td>97/424 (23%)</td>
<td>418 /515 (81%)</td>
</tr>
<tr>
<td>KSpread</td>
<td>649 /2389 (27%)</td>
<td>681 /2389 (29%)</td>
<td>138/681 (21%)</td>
<td>511 /649 (79%)</td>
</tr>
<tr>
<td>quantLib</td>
<td>549 /2701 (20%)</td>
<td>924 /2701 (34%)</td>
<td>112/549 (12%)</td>
<td>434 /549 (80%)</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>3404/11438 (30%)</td>
<td>2269/11438 (20%)</td>
<td>414/3404( 18%)</td>
<td>2990/3404(88%)</td>
</tr>
<tr>
<td>Avg.</td>
<td>27%</td>
<td>28%</td>
<td>19%</td>
<td>82%</td>
</tr>
</tbody>
</table>
These results are supportive of [Raghavan et al. 2004] where they found that most bug fixes involve the modification of a loop or changes to the condition of an if-statement. Another study [Hindle, German, Holt 2008] found that small commits are more corrective. Our results also support these findings. As we discussed before, commits with design changes are larger and most of them are not bug fixes (i.e. not small).

Revision 688568 on KSpread is an example of the small changes (files and lines) related to bug fixes. Specifically, this revision changed the code by adding an if statement to check a null pointer. The revision is identified as a bug fix because of the keyword “fix” that appeared in the commit message. The commit message of this revision is “Style Fix crash. The pointer may be null.” The revision only updated one file (Style.cpp) by deleting one line and adding two. The following is the code change.

```
- keysToStore = difference(*namedStyle);
+ if (namedStyle)
+   keysToStore = difference(*namedStyle);
```

Revision 690657 is another example of a bug fix. The commit message is “fix fontsize in dialog cell format”. This revision updated only one file (LayoutDialog.cpp) and has the following code change.

```
- selFont.setPointSize(size_string.toInt());
+ if (size_string.toInt()>0)
+   selFont.setPointSize (size_string.toInt());
```
Of course not all bug fixes are completely free of design changes. In some cases, adding a design element (e.g. new method) could be a result of fixing a bug caused by the absence of certain functionality, more along the lines of a new feature. For example, the revision 875563 on Kate is a bug fix. The commit message is “Make incremental search bar care about the user's manual cursor position changes BUG: 173284”. This commit also impacted the design of Kate by adding the method \texttt{KateSearchBar::onCursorPositionChanged}.

The last column of Table 3-8 (Not Design \(\cap\) Bug fix) can be used to address the following research question. \textit{Can we use a categorization of commits based on impact to design to determine if the commit is a bug fix?} In other words, can we say that if the commit does not impact the design then it is a bug fix? This could be used as an alternative (or better way) to identify bug fixes than analyzing commit messages. Commit messages are usually incomplete and developers may not always follow the committing standards regarding commit messages.

In Kate, there are 418 commits that are bug fixes and have no design impact. So, 81\% of the bug fixes are reported among the design commits. In the average of the four projects, 82\% of the bug fixes commits are reported (covered) within the commits that have no design impact. This shows us that a majority of the commits with no design impact are bug fixes.

3.6 Commit Labeling

One direct application of identifying design changes caused by commits addresses that fact that a commit message tells why the code is changed but often does not tell what
exactly changed in the API. An example taken from the Kate repository (revision 724732) includes the (exact) commit message “remove wrong signal declaration, obviously i’m STUPID and shouldn’t be writing code at all”. Unfortunately, this commit only tells us what file was changed; the commit message does not tell us what was removed. In actuality, the developer removed the method `KateView::cursorPositionChanged` from the header file `KDE/kdelibs/kate/view/kateview.h`. Using our approach we can automatically label commits and provide more useful information about the change.

Table 3-9 shows a sample of such labels. The first column is the revision number. The second column represents the type of the design change. The addition and the subtraction symbols indicate whether the design change has been added or deleted by the revision. The column Impact shows the actual design. So in revision 728467 a new dependency relationship was added between classes `KateView` and `KAction`. This labeling can be extracted from an existing revision system from the revision number and the files that were part of the commit.

**Table 3-9 An example of labeling commits that have design impact on Kate**

<table>
<thead>
<tr>
<th>Revision</th>
<th>Impact Type</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>513128</td>
<td>- Class DOMFunction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Method DOMFunction::InternalFunctionI</td>
<td></td>
</tr>
<tr>
<td>728467</td>
<td>+ Dependency <code>From KateView To KAction</code></td>
<td></td>
</tr>
<tr>
<td>727168</td>
<td>- Method <code>KateModeManager::reverse</code></td>
<td></td>
</tr>
</tbody>
</table>

This labeling directly supports maintenance activities. At any time developers can locate commits that have affected the design in specific way in case there is a need to
study and analyze these commits. So, this labeling scheme could serve as a legend to the evolutionary history of the system, more specifically design evolutionary history.

3.7 Threats to Validity

For the empirical study there are some possible threats to the results. One major threat is refactorings. The srcTracer tool does not deal with all variations for renames (only class rename). Another threat is that we used a lightweight analysis for the commit message to identify bug fixes. There are some bug fixes that can’t be identified using this approach and some developers do not follow the committing guidelines.

3.8 Related Work

The related work mainly focuses on research that has been done on commit analysis and categorization. The most closely related works are discussed.

Fluri and Gall [Fluri, Gall 2006] developed an approach for analyzing and classifying change types based on code revisions. Their taxonomy of source code changes reflects the significance level of a change. They define significance as the impact of the change on other source code entities. Our work differs in the method of identifying code changes and in the level of granularity. They focused on the class and methods internal code changes while we focused on higher level code changes that impact design, mainly relationships. Alali et al. [Alali, Kagdi, Maletic 2008] examined the version histories of nine open source software systems to uncover trends and characteristics of how developers commit source code. They found that approximately 75% of commits are quite small. A similar study is also done by Hattori and Lanza
[Hattori, Lanza 2008]. They provided another classification for commits according to the
development activity. They used keywords in comments as classification criteria. Hindle et al. [Hindle, German, Holt 2008] analyzed and provided a taxonomy for large commits.
Their goal was to understand what prompts a large commit. They showed that large
commits are more perfective while small commits are more corrective. German [German
2004] studied the characteristics of a Modification request (MR). Some main
observations were that bugMRs contain few files and commentMRs contain a large
number of files. Beyer and Noack [Beyer, Noack 2005] introduced a clustering method,
using co-change graphs, to identify clusters of artifacts that are frequently changed
together. Xing and Stroulia [Xing, Stroulia 2004a] provided a taxonomy of class-
evolution profiles. They categorized classes into eight types according to their
evolutionary activities.

Purushothaman and Perry [Purushothaman, Perry 2005] studied the properties and
the impact of small code changes. They found that there is less than 4 percent probability
that a one-line change will introduce a fault in the code. Ratzinger et al. [Ratzinger,
Sigmund, Vorburger, Gall 2007] used data from versioning systems to predict
refactorings in software projects. Their mining technique does not include source code
changes. The approach has been used in [Ratzinger, Sigmund, Gall 2008] to analyze the
influence of evolution activities, mainly refactoring, on software defects. Fluri [Fluri,
Wursch, Gall 2007] described an approach to map source code entities to comments in
the code. They also provided a technique to extract comment changes over the history of
a software project. A comprehensive literature survey on approaches for mining software
repositories in the context of software evolution is presented by Kagdi et al. in [Kagdi, Collard, Maletic 2007b].

In the literature, there is a lack in studies that empirically investigate the impact and the characteristics of commits, in open source projects, on design.

3.9 Summary

A detailed empirical study that used srcTracer for automatically identifying commits that impact design and break code-design consistency to understand the evolution of four software systems was presented. It was observed that only a small number of changes impact the design and most of these changes are API changes (i.e. methods). It is also shown that most design changes of the commits are small and few commits have big impact on design.

Additionally, it was found that commits with design changes contain more files and significantly more changed lines. The observations also indicate that most bug fixes do not impact the design.
CHAPTER 4

Measuring Class Importance in Code-to-Design Inconsistency

This chapter addresses one direct application for identifying design changes from revisions. Some classes have a major role in breaking code-to-design traceability and reshaping the design. It is necessary to identify classes that were involved in most of design changes. It would help in bringing more attention to these classes that are critical to preserving code-design consistency. So, these key classes must be carefully updated and documented to avoid major code-design inconsistencies and to keep the design well organized and intact during evolution.

Classes that are often impacted by design changes are branded as important concepts in the system. Identifying these important classes helps reveal what parts of the system are regularly evolved (e.g., specific features or cross cutting concepts).

The importance of a class in design is identified by how often it impacts during design changes over time. For a class A, number of revisions with design changes that involved class A is used to measure the importance of class A (*importance of a single class*). We are also interested in identifying sets of collaborating classes that are impacted together in the context of design evolution and measure their importance (*importance of multiple classes*). These sets represent classes, which co-evolve, and participate together to realize a feature or concern. This extends the idea of single important classes of the design to sets of classes that when taken as a group, are critical to the design.
Two techniques are used to automatically identify collaborative classes and measure their importance to the design. These techniques are itemset mining and agglomerative hierarchical clustering. Both approaches have been applied to the version histories of two open source projects and the results are compared.

Two out of the four open source projects discussed in Chapter 3 are used as a case study to illustrate and validate the approach. These projects are Kate and KSpread. The same revisions and extracted information are used.

Parts of this chapter are published in the 18th IEEE International Conference on Program Comprehension (ICPC’10) [Hammad, Collard, Maletic 2010b]. Section 4.1 briefly describes the case study. Measuring importance of single classes is detailed in Section 4.2. Identifying collaborative classes and measuring their importance are detailed in Section 4.3. The evaluation is presented and discussed in Section 4.4. Section 4.5 details the related work.

4.1 Classes Impacted by Design Changes

The same collected data for the case study in Chapter 3 (Section 3.1) for Kate and KSpread was used. All commits for three years time duration have been extracted. Commits with no C++ code changed are filtered out.

Only the classes involved in design changes are of interest. Therefore, only the commits that involved design changes were studied. After extracting all commits for three years time duration, code changes in these commits are analyzed by the tool srcTracer. Based on this analysis, commits are categorized as commits with design changes (impact) and commits with no design changes.
For example, the following design changes are reported by srcTracer after analyzing the code change of revision 502467 on Kate.

NEW METHOD KateView::document
NEW METHOD KateDocument::ignoreModifiedOnDiskOnce
NEW DEPENDENCY FROM KateView TO KateDocument
OLD DEPENDENCY FROM KateDocument TO KateViewInternal

The names of all classes appeared in these changes (in bold) are extracted (union). So, revision 502467 will be considered as a design impact (change) that involved classes \{KateViewInternal, KateView, KateDocument\}. From the data mining field point of view, the commit is a transaction with three items (i.e. classes).

For the program Kate, the total number of transactions was 424, i.e., 424 commits included design changes. The total number of distinct items (classes) is 334. For KSpread, there are 681 commits with design changes and 542 unique classes involved in these design changes. It is important to point out that only classes with design changes are included. Using this approach, we focus on investigating classes that are responsible for high-level changes (e.g. features). Classes with only changes that do not affect the design (e.g. fixing loops, adding comments, etc.) are ignored. Table 4-1 summarizes the extracted and collected data for Kate and KSpread.

Table 4-1 Information about the extracted commits from Kate and KSpread

<table>
<thead>
<tr>
<th>Project</th>
<th>Directory</th>
<th>Total Commits</th>
<th>Design Impact</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>KDE/kdelibs/kate</td>
<td>1592</td>
<td>424/1592</td>
<td>343</td>
</tr>
<tr>
<td>KSpread</td>
<td>koffice/kspread</td>
<td>2389</td>
<td>681/2389</td>
<td>542</td>
</tr>
</tbody>
</table>
4.2 Importance of Individual Classes

The goal is to investigate the importance of individual classes in design changes. Based on this measure we would identify key and important classes in the system. We would also identify which part of the system was heavily evolved. Identifying these classes would help in recognizing the evolved features and concepts during a time duration.

The approach is to measure class importance in design changes based on the number of commits with design changes that impacted the class. The approach of measuring importance of a single class A is summarized by the following steps.

1. Extract all commits in specific time ranges from the software repository
2. Find set of commits \( (S) \) that impacted the design by analyzing code change using srcTracer tool
3. Find subset of commits \( (Sub) \) that impacted class A from set \( (S) \)
4. Importance of class A is the cardinality of \( (Sub) \)

The number of design-change commits that a particular class is involved in measures the importance of that class in design changes. If most of the commits with design impact included class A, then this class is very important to the design. From a higher level point of view, the feature(s) related to this class is heavily updated and evolved. So the number of commits per class, in a specific time duration, gives valuable information about the importance of the class in design and the specific feature(s) that are evolved (added, deleted, or update).
Based on this measure, Table 4-2 shows the ten most important classes on designs of Kate and KSpread. Classes are ranked according to their importance which is the number of commits (#Commits). The most impacted class by Kate’s design changes is KateView. For KSpread it is class Sheet. During the studied three years, 81 revisions committed design changes on Kate where KateView was part of these design changes. These 81 revisions are out of the total number of commits (424) with design impact. So, the importance of KateView, compared to all other classes, is 19% (81/424).

The importance of KateView (ranked 1) shows that it was the core of most design changes on Kate during the studied three years. It also means also that the features related to KateView have been considerably evolved. The same applies for class Sheet in KSpread.

Table 4-2 Top 10 classes with highest importance on Kate and KSpread.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>#Commits (Importance)</th>
<th>Kate</th>
<th>KSpread</th>
<th>Kate</th>
<th>KSpread</th>
</tr>
</thead>
<tbody>
<tr>
<td>KateView</td>
<td>81(19%)</td>
<td>178(32%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateDocument</td>
<td>72(17%)</td>
<td>148(27%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateViNormalMode</td>
<td>49(12%)</td>
<td>106(19%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateSearchBar</td>
<td>34(8%)</td>
<td>97(18%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateViewInternal</td>
<td>33(8%)</td>
<td>80(15%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateGlobal</td>
<td>31(7%)</td>
<td>58(11%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateCompletionWidget</td>
<td>27(6%)</td>
<td>51(9%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateDocumentConfig</td>
<td>18(4%)</td>
<td>42(8%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateAutoIndent</td>
<td>16(4%)</td>
<td>41(7%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KateTextLine</td>
<td>16(4%)</td>
<td>41(7%)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Most classes were involved in very few design changes (small number of commits). Only a few classes were impacted by most of design changes. The distribution of classes, involved in design changes during the studied three years, based
on their importance for the four projects is shown in Table 4-3. Classes are grouped into four importance ranges (values). The columns are the importance values. These values are divided into four fixed ranges (0-1, 2-3, 4-6 and greater than 6). For Kate, 114 classes of all the 334 classes involved in design changes (34%) were impacted by only one commit. For 36% of the classes, their importance values are 2 or 3 commits. The percentage of classes that participated in 4, 5, or 6 commits is 17%. Only 13% of classes have been impacted by more than 6 commits. In average, for the two projects 67% of classes have been impacted by only 1, 2, or 3 commits which means most class are not that important (low importance value).

Table 4-3 Distribution of classes based on their importance for Kate and KSpread

<table>
<thead>
<tr>
<th></th>
<th>Class Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>1</td>
</tr>
<tr>
<td>Kate</td>
<td>34%</td>
</tr>
<tr>
<td>KSpread</td>
<td>28%</td>
</tr>
<tr>
<td>Avg.</td>
<td>33%</td>
</tr>
</tbody>
</table>

Figure 4—1 shows the word cloud (by www.wordle.net) for Kate and KSpread classes based on their importance to the design (number of commits). In word cloud view, the size of the word is proportional to its occurrences in the text or article. For the word cloud shown Figure 4—1, the size of the word (class) is proportional to the number of commits that impacted the class (importance). Most of the classes look very small because their importance is minimal. This cloud view of classes helps to easily identify which part of the design has been mostly evolved and would give a hit about evolved
features and concepts related to these classes. Classes with largest font sizes are the most important classes in design changes (e.g. KateView and KateDocument, and Sheet).

![Word cloud view for classes involved in design changes for (A) Kate and (B) KSpread. Size of the word (class) reflects the importance](image)

**Figure 4—1** Word cloud view for classes involved in design changes for (A) Kate and (B) KSpread. Size of the word (class) reflects the importance

### 4.3 Importance of Multiple Classes

Section 4.2 discussed measuring importance of single classes. Now we need to measure the importance of *multiple* classes that usually collaborate together to implement a feature or concept. Multiple classes usually collaborate to implement a feature or to achieve a high-level specific task (e.g. cross-cutting concern). For example, to implement the feature (class) Employee, many classes may involve (e.g. BankAccount
and *Address*) in implementing this feature. These design changes may involve these classes directly or indirectly. In a direct design change the class was modified. In an indirect design change the relationships to the class were modified. This indirect involvement in a design change is not the same as finding co-changed classes. Only the class *Employee* may have undergone change while the classes *BankAccount* and *Address* have not. This is especially the case for when new relationships are formed to existing classes. Even though only the class *Employee* has changed, all the three classes collaborated (i.e. appeared together) to implement the feature involving the *Employee*. We call this set *collaborative classes*. Specifically, two classes A and B are collaborative if:

- Class A added/deleted relationship to B and vice versa (B to A)
- Class A and B appeared together in a code change on the file level

Figure 4—2 shows an example of collaborative classes and is a portion of the code changes from the file *test_regression.h*. This code change is portion of changes committed by revision 589090 in Kate. Form the commit message; this revision added some features including the feature “*added output-object for direct manipulation of result file*”. To implement this feature, revision 589090 impacted the design by adding a new class *OutputObject* with a set of methods. Code change to the class *OutputObject* also indirectly involved the classes *KateView, KateDocument* and *KJS::JSObject*. As a result all three classes collaborate to achieve a specific high-level goal which is adding the feature “output an object”.
It is important to point out that classes \textit{KateView}, \textit{KateDocument} and \textit{KJS::JSObject} are not necessarily changed. The goal is not to identify co-changed classes, but the goal is to identify collaborative classes that were involved together in design changes. By including only classes with design changes, we attempt to identify collaborative classes that are mostly responsible for a feature, concept or cross cutting concern.

\begin{figure}
\begin{verbatim}
+class OutputObject:public KJS::JSObject
+{
+  public:
+    OutputObject(KJS::ExecState *exec, KateDocument *d, KateView *v);
+    virtual ~OutputObject();
+    virtual KJS::UString className()const;
+    void setChangedFlag(bool *flag) { changed = flag; }
+    void setOutputFile (const QString &filename)
+    {this->filename = filename; }
+  private:
+    KateDocument *doc;
+    KateView *view;
+    bool *changed;
+    QString filename;
+    friend class OutputFunction;
+};
\end{verbatim}
\caption{Example of code change with collaborative classes that impacted together by design changes}
\end{figure}

Two different data mining techniques are presented and discussed to identify collaborative classes and measure their importance. The first technique is itemset mining and the second technique is the agglomerative hierarchal clustering. We discuss each technique and show the benefits that can be gained from each one.
4.3.1 Itemset Mining

The first method that we discuss to identify collaborative classes and measure their importance is itemset mining [Agrawal, Srikant 1994]. The basic idea is that all classes that have been impacted by design changes in the same commit are likely to be collaborative. So, the goal is to identify frequent sets of classes, with different sizes, that impacted by the same commit.

Each commit with design impact is represented as a transaction of classes (items). All classes that impacted the design in the same commit are assumed to be potential collaborative classes. Then we applied the Apriori algorithm [Agrawal, Srikant 1994] to all transactions. The Apriori algorithm is well known in the data mining field to identify frequent sets of items that appear together (frequent).

Apriori algorithm is a direct technique to identify sets of collaborative classes, with different sizes, and measure their importance. The method is summarized as follows.

1. Represent each commit with design impact as a transaction of items (classes)
2. Apply Apriori on these transactions with appropriate support value
3. Consider frequent itemsets as collaborative classes and frequencies as their design importance.

The Apriori algorithm steps are summarized in the following procedure.

1. \( L_1 = \{ \text{frequent items with length 1} \} \)
2. For \((k = 2; L_{k-1} \neq \emptyset; k++)\) Do
   - Generate candidate itemsets \( C_k \) from frequent \( L_{k-1} \) itemsets
- If support of \( C_k < \text{min}\_\text{support} \) then eliminate \( C_k \) (not frequent).
  
  else add \( C_k \) to the answer set (frequent).

The first step is finding frequent items with size 1. Frequent means greater than or equal support. The second step is the process of generating itemsets with size \( k \) from frequent itemsets of size \( k-1 \). Then the frequency of new generated itemsets is counted by scanning the database. The process continues until no more frequent itemsets (with size \( k \)) can be generated from frequent itemsets of size \( k-1 \).

Table 4-4 shows results about the identified itemsets for the two projects. The support values were three and five for Kate and KSpread respectively. The support values were selected to keep the number of itemsets reasonable (hundreds not thousands).

Applying Apriori algorithm on transactions of both projects, 392 itemsets were identified in Kate and 353 itemsets in KSpread. The distribution of these sets based on size is shown in Table 4-4.

### Table 4-4 Results of the identified itemsets by Apriori for Kate (support=3) and KSpread (support=5)

<table>
<thead>
<tr>
<th></th>
<th>Kate - Support = 3</th>
<th>KSpread - Support = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Size</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Number of sets</td>
<td>193</td>
<td>131</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>23</td>
<td>8</td>
</tr>
<tr>
<td>Set Size</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Number of sets</td>
<td>188</td>
<td>116</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>57</td>
<td>22</td>
</tr>
</tbody>
</table>

For Kate, the size range of the identified itemsets (with support value 3) is from two to six (see Set Size). The “Number of Sets” is the number of identified sets in each
category. For example, there are 132 identified itemsets with size three (i.e., three collaborative classes). For all sets in each category, the frequencies are computed as the number of commits (transactions). Maximum frequency values are reported in (Max frequency). In Kate there is at least one set with size three that has eight frequencies and no other set within the same category (size 3) that has more than eight frequencies.

For example, Table 4-5 shows the most frequent sets with sizes 2-6 from Kate. The three classes KateViewInternal, KateView and KateViNormalMode were impacted together eight times. So, their importance together is eight.

Classes in each itemset are collaborative. So, we have sets of collaborative classes with different size (2 to 6) in Kate and (2 to 5) in KSpread. The frequency of each itemsets is the importance of collaborative classes together. For example the importance of classes KateDocument and KateView together is 24 (maximum frequency). This means among all pair-wise classes, these two classes together have the highest influence on design changes during the studied three years.

**Table 4-5 Some identified itemsets (collaborative classes) from Kate by Apriori**

<table>
<thead>
<tr>
<th>Set (Collaborative Classes)</th>
<th>Set Size</th>
<th>Frequency (Importance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KateDocument KateView</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>KateViewInternal KateView KateViNormalMode</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>KateCmdActionItem KDialo kateCmdAction KateCmdActionMenu</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Date Character SedReplace SearchCommand CoreCommands</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>QTreeView KateArgumentHintTree KateArgumentHintModel ExpandingWidgetModel KateCompletionDelegate KateCompletionModel</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
The frequency of itemsets with size one (1-itemset) is the measure that we discussed in Section 4.2 (importance of individual classes).

4.3.2 Hierarchical Clustering

Another method to identify collaborative classes with their importance is clustering. The goal of clustering is to group commits with design impact into disjoint (similar) groups (or clusters). Commits that have similar impacts on classes are grouped together. As a result, we will have groups of collaborative classes in each cluster. The method is summarized as follows.

1. Represent each commit with design changes as vector of keywords (classes)
2. Apply Agglomerative hierarchical clustering [Johnson 1967] of these vectors with appropriate similarity value
3. Consider the intersection of each resulted cluster as collaborative classes and size of the cluster as their design importance.

Usually features and concepts are implemented (or maintained) by more than one commit that impact specific and small parts of the design (same classes). So, common classes in a cluster would be collaborative.

The same class representation (as itemset mining) is used for clustering. Each commit with design impact is represented by a vector of keywords (classes). Similarity is then measured among these vectors. Collaborative classes can be identified from the common classes (intersections) among vectors (commits) in each cluster. Using clustering, only very similar commits are grouped together in one cluster, which would mean related programming (design) activities.
Another advantage is the outcome of the clustering algorithm. Because we have chosen the hierarchal clustering algorithm, the outcome is disjoint clusters of commits. This could be a first step toward recovering traceability links among commits and identifying commits that are responsible for implementing specific feature or concept. So, the archive history (commits) is partitioned into high-level related maintenance topics or activities.

Once each commit is transformed into a vector of keywords (i.e. classes), clustering can be easily applied. Agglomerative hierarchal clustering is a bottom up clustering technique. It starts by finding the most similar pairs and group them into 2-elements (or 1-element) clusters. Next, the similarity between clusters from the previous step is computed and new larger clusters are generated. The process continues in a hierarchal way until we end up with one cluster that contains all elements. The similarity between clusters decreases while going up in the hierarchy. The generated hierarchal structure is called a dendrogram. So, a cut is done at a similarity threshold value in the dendrogram.

*Jaccard similarity* coefficient measure has been used to calculate similarities between vectors of classes (commits). For two vectors of keywords $V_1$ and $V_2$, the similarity is measured by the number of common elements (classes) between the two vectors (intersections) divided by the number of unique elements in both vectors (union).

\[
\text{Similarity} = \frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}.
\]
For example if vector V1 contains the three elements \{A, B, C\} and vector V2 contains the three elements \{B, C, E\}, the similarity between V1 and V2 is 50%: \(|\{B, C\}| / |\{A, B, C, E\}| = 2/4 = 0.5\).

Figure 4—3 Portion of dendrogram resulted by applying hierarchal clustering on commits of Kate shows portion of the hierarchal clustering output (dendrogram) of revisions on Kate. In the first step revisions 574827 and 580338 are grouped in one cluster (C1) because they are identical since 2/2 = 1 (100%). Both revisions have the same classes (KateScriptIndent and KateIndentJScript). Revisions 738455 and 859278 are also grouped (C2) in the first step because they are identical too. In the next step (level), a new cluster (C3) is generated from cluster C1 and revision 580389. The similarity between C1 and revision 580389 is 70%. Revision 580389 has only one extra class (KateNormalIndent) that does not exist in any revision in C1 (2/3 = 0.7). In the next level, a new cluster (C4) is generated from C3 and revision 814327. The similarity is 20% (1/5 = 0.2). Revision 814327 differs in two classes (KateJSIndenterFun and KateJSIndenter) from revision 580389. Finally in the last level, all elements are grouped in one cluster with the minimum similarity value.

Based on a similarity threshold value, we can make a cut in the hierarchy. The number of clusters varies based on the cut point. But all clusters are disjoint. We set the threshold value to be 50%. All clusters with similarity value greater than 0.5 are considered similar. In this way we at least guarantee that revisions in each cluster are mostly similar.
Table 4-6 shows some statistics about the identified clusters after applying hierarchal clustering on the revisions of Kate and KSpread. For Kate, there are 424 commits that have been clustered into 60 disjoint clusters of revisions. The average size of these 60 clusters is 3.6 commit per each cluster. The maximum size for all the 60 clusters is 25 commits.

Table 4-6 Information about identified clusters by hierarchal clustering for Kate and KSpread

<table>
<thead>
<tr>
<th>Project</th>
<th>#Commits</th>
<th>#Clusters</th>
<th>Avg. Size</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td>424</td>
<td>60</td>
<td>3.6</td>
<td>25</td>
</tr>
<tr>
<td>KSpread</td>
<td>681</td>
<td>90</td>
<td>3.3</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 4-7 shows a sample from the identified clusters from Kate. The (Cluster) is the identified clusters with their elements (revisions). The (Cluster Size) is the number of revisions per cluster which is the importance of the cluster. The intersection is the
common classes in all revisions within each cluster (collaborative classes). For example, one cluster has grouped five revisions (852305, 852669, 846633, 847933, and 846631). All these revisions have impacted classes KateViNormalMode and KateViVisualMode.

Table 4-7 Some Identified Clusters of commits by the hierarchal clustering on Kate

<table>
<thead>
<tr>
<th>Clusters</th>
<th>Cluster Size (Importance)</th>
<th>Intersection (Collaborative Classes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>847131 848793 848794</td>
<td>3</td>
<td>KateViInsertMode</td>
</tr>
<tr>
<td>852305 852669 846633</td>
<td>5</td>
<td>KateViNormalMode KateViVisualMode</td>
</tr>
<tr>
<td>847933 846631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>806390 814327</td>
<td>2</td>
<td>QItemDelegate ExpandingDelegate QStyledItemDelegate</td>
</tr>
<tr>
<td>868385 872168 738455</td>
<td>5</td>
<td>KateView KateViewInternal</td>
</tr>
<tr>
<td>859278 502467</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The intersection of revisions in a cluster (common classes) is used as another criterion to identify collaborative classes. This method is closely related to the itemset mining but it has the advantage of computing similarity. This technique takes into consideration the impact of the design change per commit. Classes KateViNormalMode and KateViVisualMode were impacted together by five revisions, hence their importance together is five (cluster size).

Each cluster of commits could be seen as a feature or concept related activities. Common classes in each cluster would represent sets of related features. Commits are the activities that added, deleted, or maintained these features or concepts.

Itemset mining has the advantage of exactly measuring the importance of more than one class. Hierarchal clustering has the advantage of identifying similar commits and groups them in disjoint clusters.
4.4 Evaluation

The evaluation aims to verify that the identified collaborative classes, by the itemset mining and the hierarchal clustering, have collaborated together in code changes. The verification is done for each set of classes. All classes under verification must appear together, at least once, in a code change (i.e. hunk). The scope of the code change (hunk) is the file.

As shown before in Figure 4—2, based on our definition of collaborative classes, they must participate together in code change.

We automatically verified all code changes of commits with design impacts. For each set of classes, we check whether all classes participated in code changes on the file scope. Based on this analysis, sets of classes can be verified as related or not related. For an identified set of classes $S$, the following evaluation procedure is applied.

1. For each changed file by commits with design impact do:
   a. Extract code changes
   b. If all classes of $S$ found in code changes then Report verified and Exit
   c. Else repeat step 2

2. Report not collaborated

The following two subsections discuss the evaluation results of identified sets of classes by the Apriori algorithm and the hierarchal clustering.

4.4.1 Itemset Mining

We verified each set identified by the Apriori algorithm. We evaluated and compared the results of different support values for each project. The goal is to identify
this appropriate support value that minimizes false positive results. We want to get high recall (more identified) and high precision (more verified as collaborative). We found that the appropriate support value is the one that generate some hundreds item sets not thousands.

For each project we evaluated generated item sets with different support values. Low support value increases recall while high support value increases precision. Table 4-8 shows the evaluation results for identified sets for Kate and KSpread for different support values. The number of identified sets of classes for each support value is shown in (Identified).

**Table 4-8 Evaluation results for identified item sets for Kate and KSpread with different Support values**

<table>
<thead>
<tr>
<th>Support</th>
<th>Kate</th>
<th>KSpread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>24911</td>
<td>392</td>
</tr>
<tr>
<td>Verified</td>
<td>7135</td>
<td>306</td>
</tr>
<tr>
<td>Accuracy</td>
<td>29%</td>
<td>78%</td>
</tr>
<tr>
<td>Max Verified Set Size</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

The number of correctly verified sets of classes as collaborative by inspecting code changes is shown in (Verified). For Kate the support values are two, three and four. For KSpread support values four and five were used (more items and transactions than Kate). The size of maximum verified set (collaborative classes) for each support value is shown in (Max Verified Set Size).
For Kate, 24911 sets of classes were generated when the support value is two. This is the maximum identified sets (recall) that can be generated (at least two collaborative classes). Only 7135 (29%) of these sets of classes were verified as collaborative (low accuracy). With support value three, the accuracy is improved to 78% but the recall is decreased from 7135 (support 2) to 306. The same pattern follows with higher support values. The same applies for KSpread. With support four we get high recall (736) but low accuracy (2%). The accuracy improves to 99% with the support value five and the recall decreases to 349. The detailed evaluation results about each support value are shown in APPENDIX A.

The best results are reported (high recall and precision) with support values three for Kate and four for KSpread. These support values keep the number generated itemsets reasonable (hundreds not thousands).

4.4.2 Clusters

For each cluster we verified classes that are common in all commits (intersection). The verification was applied for clusters which have more than one class in common. Table 4-9 shows the number of identified sets of classes (intersection of each cluster) for the two projects in (Identified). For example, in KSpread there are 66 sets of classes identified from the intersection of each cluster of revisions. The number of verified sets among these 66 sets is 56 (85%).

The sizes of the verified sets of classes (collaborative) from Kate are (2, 3, 4, 5, and 10) and (2, 3, 4, 5, 6, and 7) for KSpread. The detailed evaluation results about clusters are shown in APPENDIX A.
It is found that 35% of the clusters in Kate have one developer per cluster. For Kate also, 12% of the clusters have one developer for more than half of the commits. For KSpread, we found that 70% of the clusters have one developer per cluster. These observations are indication that these clusters have related activities or expertise.

Table 4-9 Evaluation results for identified clusters for Kate and KSpread (low recall and high precision)

<table>
<thead>
<tr>
<th></th>
<th>Kate</th>
<th>KSpread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>37</td>
<td>66</td>
</tr>
<tr>
<td>Verified</td>
<td>33</td>
<td>56</td>
</tr>
<tr>
<td>Accuracy</td>
<td>89%</td>
<td>85%</td>
</tr>
<tr>
<td>Max Verified Cluster Size</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

4.4.3 Variability of Clusters

The goal of this section is to measure the gaps between revisions in a cluster. For example, if there are 20 revisions with a design impact, the gap between the first and the fifth revisions is 3 and the gap between the 9th and the 10th revisions is zero (consecutive revisions). This measure is another indication of the quality of produced clusters. Consecutive or very closely spaced revisions in a cluster could be an indication that the commits have related implementation activates (specific feature or concept).

All commits participated in the clustering algorithm were given a sequential number according to the date. So, all commits on Kate were given sequence numbers from 1 to 424. For KSpread, commits were numbers from 1 to 339. For example, the first revision with design change on KSpread is 493943 on the first of March 2006 was
numbered 1 and the last revision 890998 on the 30\textsuperscript{th} of November 2008 was numbered 681 (total commits with design changes). Another example is the cluster of revisions (847131, 848793, and 848794) on Kate was numbered (345, 354, and 355).

We used a simple measure, from statistics, for variability which is the range. Range is the difference between the maximum and the minimum value in a set. The Range value of cluster (345, 354, and 355) is 10 (355 - 345). To make the value more meaningful to our data representation (i.e. counting gaps), we modified the Range by subtracting the number of revisions in the cluster minus one from it.

\[ \text{Variability} = [\text{Max(cluster)} - \text{Min(cluster)}] - \text{size(cluster)} - 1 \]

Using this measure, the variability value for consecutives numbers will be zero. For example, the sequence 1, 2, 3, and 4 will be zero (4-1-3), which means the revisions are consecutives (not variant). For the cluster (345, 354, and 355), the variability value is eight, which means there are eight revisions between 345 and 354 and zero revision between 354 and 355. Specifically, this measure counts how many gaps (revisions) among the revisions of the cluster. Table 4-10 shows the variability results for all clusters in both projects. The variability is given in ranges and clusters are distributed based on these ranges. Based on this preliminary distribution, clusters can be interpreted as:

\begin{itemize}
  \item 15\% of clusters are very close (have consecutives commits)
  \item 23\% of clusters are close
  \item 10\% are somehow close
  \item 51\% are not close
\end{itemize}
The distribution shows that small number of clusters (15%) corresponds to a specific feature or concept. About half of the clusters have large gaps and this is an indication of periodic maintenance activities on one or more features.

**Table 4-10 Distribution of clusters based on the Variability value for Kate and KSpread**

<table>
<thead>
<tr>
<th>Project</th>
<th>Variability</th>
<th>0</th>
<th>1-10</th>
<th>11-30</th>
<th>&gt;30</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kate</td>
<td></td>
<td>8%</td>
<td>27%</td>
<td>12%</td>
<td>53%</td>
<td>100%(60)</td>
</tr>
<tr>
<td>KSpread</td>
<td></td>
<td>21%</td>
<td>18%</td>
<td>8%</td>
<td>48%</td>
<td>100%(90)</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td>15%</td>
<td>23%</td>
<td>10%</td>
<td>51%</td>
<td>-</td>
</tr>
</tbody>
</table>

### 4.4.4 Itemset Mining vs. Clustering

Itemset mining method has high recall compared with clustering. On the other hand clustering can produce high precision. Thus Itemset mining method can give high precision if an appropriate support value is used. The appropriate support value which produces hundreds of sets not thousands (according to or study).

Hierarchical clustering could be used to cluster commits into disjoint related activities or topics that are responsible for maintaining a feature or concept. Item set method has the advantage of identifying more collaborative classes and measures their importance (frequency).

### 4.4.5 Threats to Validity

The results are based on applying srcTracer, to identify design, which has its own thread to validity. The case study is limited to two open source projects with one continuous time period for each project. Results may vary for other projects or variable
and different time durations. However, the extracted data and time periods are reasonable and they are not small or limited.

Fixed similarity threshold was used for the clustering. Also, specific similarity measure was used. Clustering results vary based on these two issues.

The evaluation is done automatically which may differ if done manually by experts. From a designer’s point of view, some classes are not necessarily related even if they participate in a code change. Actually, it is very hard to manually verify hundreds or thousands of classes on large code changes.

4.5 Related Work

Zaidman and Demeyer [Zaidman, Demeyer 2008] proposed a technique to identify the most important (key) classes in a system. They defined key classes as classes with a lot of control within the application (controlling functions). The degree of dynamic coupling of classes was used to identify the key classes. The approach is applied for a static, single version of the system. Their definition for key classes does not involve the design evolution and differs from our definition. Xing and Stroulia [Xing, Stroulia 2004b] used their UML differencing algorithm UMLDiff [Xing, Stroulia 2005b] to provide a class evolution taxonomy that consists of eight distinct evolution types. They also employ the Apriori association-rule mining algorithm to discover co-evolving classes. Our work differs in applying Apriori mining to classes that collaborate together in code changes and are not necessarily co-changed (co-evolved). We also differ in proposing the idea of measuring class importance to design. Lanza and Ducasse [Lanza,
Ducasse 2001] presented the class blueprint visualization to visualize the internal structure of classes.

Sager et al. [Sager, Bernstein, Pinzger, Kiefer 2006] presented an approach to detect similar Java classes based upon their abstract syntax tree representations. Our work tries to identify collaborative classes which are not necessarily similar. Gîrba et al. [Gîrba et al. 2007] proposed the usage of formal concept analysis to identify groups of entities with similar properties that change in the same way and at the same time. Using predefined conditions specific change patterns are detected. For classes, the goal is to identify bad smells.

Bieman et al. [Bieman, Andrews, Yang 2003] identified and visualized classes that experience frequent changes together. They call this change-coupling between classes. The approach depends on using class-level implementation metrics and class’s relationships. Pinzger et al. [Pinzger, Gall, Fischer, Lanza 2005] visualized the evolution of source-code entity (e.g. classes and files) and relationship metrics across multiple releases. Robillard [Robillard 2005] proposed a technique based on structural dependencies to automatically discover the program elements (including classes) relevant to a change task. The method depends on static analysis of the source code and does not take into account the history of changes.

Vaucher et al. [Vaucher, Sahraoui, Vaucher 2008] proposed techniques to discover patterns of evolution in large object-oriented systems. To locate patterns, they use clustering to group together classes which change in the same manner at the same time. For every class evolution, the amount of change is represented by two values:
implementation change IC and functional changes FC. These changes are included in a vector that defines its evolution blueprint. The XMean clustering was applied to these vectors to identify similar evolutionary patterns. We do not study the internal evolutionary patterns of classes but how classes are impacted by design evolution.

Zimmermann et al. [Zimmermann, Zeller, Weissgerber, Diehl 2005] presented a prototype called ROSE that applies data mining methods to version histories in order to guide programmers along related changes. The Apriori Algorithm is used to obtain association rules from version histories. The entities included in the mining are any changes given by file and the affected programming components. Identifying co-changed lines [Zimmermann, Kim, Zeller, Jr 2006] is an examples of applying data mining techniques identify co-changed entities from software repositories. Other examples are the identification of call-usage pattern for functions [Kagdi, Collard, Maletic 2007a] and predicting code changes [Ying, Murphy, Ng, Chu-Carroll 2004]. Li and Zhou [Li, Zhou 2005] proposed PR-Miner that uses frequent itemset mining to extract general programming rules from large software code. The technique is applied only on the source code of one version.

Beyer and Noack [Beyer, Noack 2005] proposed the clustering method co-change Graphs. The goal is to identify clusters of artifacts that are frequently changed together. The co-change graph of a given version repository is an undirected graph where vertices contains all software artifacts and edges (weighted) connect co-changed artifacts. Our approach differs in the technique and the goal. We specifically consider only classes
with design impact and they are not necessary co-changed. Our goal is to cluster commits (not artifacts) based on their high-level design impact.

Vanya et al. [Vanya et al. 2008] proposed a history–driven approach to identify evolutionary set of software entities which co-evolved in the past. The approach used hierarchal clustering. The goal is the partition of a software archive by (automatically) identifying evolution-related issues. Our approach partitions commits based on design changes in order to identify collaborative classes.

Fokaefs et al. [Fokaefs, Tsantalis, Chatzigeorgiou, Sander 2009] proposed a class decomposition method using an agglomerative clustering algorithm based on the Jaccard distance between class members. The goal is to identify the “God Class” bad smell. A review about hierarchical clustering research in the context of software architecture recovery and modularization is presented in [Maqbool, Babri 2007].

Our work is distinguished by identifying key classes based on how they are impacted by design changes. We also identify collaborated classes that are not necessarily co-changed. Finally we cluster commits based on high-level design changes.

4.6 Summary

A measure has been presented for class importance in the context of design evolution. Classes that were impacted by most design changes are then branded as key classes. The proposed measure takes into consideration high-level design changes that impact UML class diagrams. These changes correspond to features and concepts in the system. Measuring importance also has been applied on classes that often collaborate in design changes. A case study has been presented on two open source projects. Results
show that very few classes are critical to the design in the context of evolution. That is only around 15% of all classes in the two systems studied were impacted by more than six design changes.

   Itemset mining and hierarchal clustering were used to automatically identify collaborative classes. Itemset mining method identifies more collaborative classes and clustering archives design changes into disjoint related activities.
CHAPTER 5

Preserving Code-to-Design Consistency during Evolution

Identifying design changes as code evolves leads to provide many useful applications and tools that directly support in preserving code-design consistency during code changes. The chapter discusses two proposed methods to support preserving the consistency between code and design during evolution. Each of these methods helps in preserving code-design consistency in different way. These methods have different applications and they can be used in different activates. The goal of the first method is to monitor developers’ activities so they can correctly change the current code to make it consistent with a target design. The second method aims to automatically update design (if needed) after each code change to keep it consistent with code. So, by monitoring the evolving of current code based on target design, code-design traceability is correctly established and preserved. On the other hand, by automatically updating design to reflect code changes, code-design traceability is kept consistent and preserved during code evolution.

5.1 Monitoring Code-Design Traceability

During software evolution, developers are involved in many maintenance tasks. It is necessary to keep developers on track with respect to current and intended design. By achieving this goal, the code-to-design traceability is kept updated and more importantly accurate. Developers lose the architectural knowledge of the system during
evolution [Feilkas, Ratiu, Jurgens 2009]. This is because developers focus on code changes more than design. As a results traceability between code and design is broken and more over they are incorrect. Based on this, we try to address a key research issue, how to visually guide developers toward the intended design while performing a perfective maintenance task? Developers need to be aware of the status of the current code-to-design traceability and which links need to be changed to reach the intended design.

Perfective maintenance tasks normally involve changes to the design of a system. Enhancing the system by adding a new feature, refactoring, or implementing a design pattern are example tasks. In many situations the requirements of the change can be expressed in the form of the end result. That is, the intended final design can be presented as a UML class diagram. Given the final design the code can be evolved to meet the target requirements.

The proposed work compares this target design with the actual design embodied by the source code in a visual presentation using UML. The notation is extended to highlight the addition and removal of classes, methods, and relationships needed to evolve the actual design into the target design. The intent is to give developers a means to monitor the progress of the maintenance task over multiple steps and commits. This integrated view (target+actual) is termed a completion diagram and describes what still needs to be completed to meet target design within the context of the current state of the source code.
Perfective tasks often only impact a small portion of the system. While our method could be applied to a large diagram we restrict ourselves here to just the key aspects of the design involved in the maintenance task. Additionally, we avoid the problem of graph comparison altogether. We determine if a code change impacts the design directly from the code and map that to the intended target. This also presents the inherent design embodied by the code and not a view presented by outdated documents. The intent is to evolve both code and design simultaneously.

Developers will benefit from methods and tools that abstract and manipulate UML diagrams. UML is in widespread use and has an intuitive mapping from design to code. UML/onion graphs [Kagdi, Maletic 2007] and UML model slicing [Kagdi, Sutton, Maletic 2005] are examples of such methods.

5.1.1 UML Completion Diagram

The completion diagram is based on a target design diagram that shows the end result of the requested maintenance task. We will first discuss the target diagram, and then show how the completion diagram is generated based on the target. The UML target diagram represents the design that is desired for a system, or the relevant parts of a system. The source code must be evolved to meet these design requirements. For the example given in Figure 5—1 the code needs to be modified so that it has the given classes and the method `getColumnShape()` in class `RootNTuple`. The resulting part of the code will implement this design with only the relationships and classes shown.

The target diagram allows the developer to state what parts of design are important to complete the task. The system may consist of other classes but they are not of direct
importance to the current maintenance task. There may also be other relationships between a class in the target design and another class that are not of great importance to the task. The object is to evolve the current code to satisfy the target UML diagram.

![Figure 5—1 A UML target diagram with the desired design.](image1)

![Figure 5—2 A UML completion diagram. To complete the maintenance tasks the code should be changed by deleting the elements in red and adding the elements in gray.](image2)
The UML completion diagram shows the progress of the maintenance task towards that of the target design. Each UML element that needs to be added is rendered in gray. Each UML element that needs to be deleted is marked in red. The example completion diagram given in Figure 5—2 was generated from the target diagram in Figure 5—1 and the actual source code of the system. The dependency relationship between RootNTuple and DataSourceException appears because it currently exists in the code. Moreover, it is colored with red because it should be deleted to reach the desired design. Both the generalization and the aggregation relationships are colored with gray because they are not yet implemented. The same applies for the method getColumnShape.

5.1.2 Using a Completion Diagram

Given the completion diagram in Figure 5—2, a programmer can easily see the current status of the code undergoing evolution and what needs to be done to complete the maintenance task. Figure 5—3 presents the results of a series of changes to the source code and how they are reflected in a completion diagram. These examples are based on design changes from the HippoDraw open source project.

First, a developer added the following line of code in the RootNTuple class. Code changes are shown in the unified diff format.

```cpp
+ typedef std::vector<RootBranch*> BranchList_t;
```

Our tool, srcTracer [Hammad, Collard, Maletic 2009], identifies and recognizes this modification to the code as a design change. Specifically, a new aggregation relationship was added between classes RootNTuple and RootBranch. Based on this, the
Completion UML diagram is updated in Figure 5—3(A) with the color of the aggregation relationship changing from gray to black.

Figure 5—3 Completion diagrams for visualizing the progress of a maintenance task for the target design in Figure 5—1. The starting completion diagram is shown in Figure 5—2. Each part shows the effect of code changes towards the completed task.
The next code change adds a new generalization between classes *RootNTuple* and *DataSource*.

```cpp
- class RootNTuple {
+ class RootNtuple : public DataSource {
```

This is reflected in the completion diagram in Figure 5—3(B) where the color of the generalization relationship is changed from gray to black. The following code change removed the only use of the class *DataSourceException* in the class *RootNTuple*.

```cpp
- throw DataSourceException ( what );
+ throw runtime_error ( what );
```

The completion diagram Figure 5—3(C) shows the elimination of this relationship.

The last code change to perform that will finish this design is the addition of the method *getColumnShape* to the class *RootNTuple*.

```cpp
+ void getColumnShape();
```

This completes the maintenance task and the completion diagram is identical to the target diagram (Figure 5—1).

### 5.1.3 Creating Completion Diagrams

This section describes the process of creating completion diagrams and shows in more detail how code changes are mapped to the visualization. As source code changes are made, the completion diagram is updated. We analyze the code changes and identify
which changes impact the design. As design changes are found they are compared to the completion diagram and updates are made as necessary.

Internally, UML diagrams are stored in our XML representation, classML. The completion diagrams are rendered and drawn based on classML. The following is a portion of the classML representation for class RootNTuple shown in Figure 5—2.

```xml
<class>
  <name> RootNTuple </name>
  <methods>
    <method> public void getColumnShape() </method>
  </methods>
  <generalizations>
    <generalization> DataSource </generalization>
  </generalizations>
  <dependencies>
    <dependency> TTree </dependency>
  </dependencies>
</class>
```

Each UML (design) entity is tagged with the name its type. For example the tag <methods>…</methods> encloses all methods of the class. Each method is also tagged with its own tag <method> …</method>.

Completion diagrams are generated from the given target diagram and actual code. The target UML diagram is represented by classML with source code is represented srcML. The general process is to add special completion stereotypes, i.e., toBeAdded and toBeDeleted, to the classML representation of the target diagram. These stereotypes are added as properties for each UML element in classML (i.e. tag). The completion diagram is rendered with each element colorized based on its completion.
The new stereotype property has been added to the elements that need to be added or deleted from the current code. The dependency relationship to `DataSourceException` has the stereotype property value “toBeDeleted”. Based on this value, the relationship in the completion diagram is rendered with red color.

Once the completion diagram is created, it is incrementally and dynamically updated based on the code changes. For, each code change, `srcTracer` is triggered to analyze this change. If the change is design change, the design change is compared to the classML representation of the completion diagram. Based on this comparison, the tag
property of the UML changed element is updated and the UML diagram is redrawn based on the updated classML.

5.1.3.1 Elements to be Added

For elements in the target diagram that need to be added, the process is straightforward. Each UML element in the completion diagram is checked to see if it exists in the current source code. If it does not exist, the completion stereotype toBeAdded is placed on the UML element. The process is summarized by the following steps:

1. Represent the target UML diagram in the XML representation classML
2. Represent the current source code in the XML representation srcML
3. Parse classML to extract all UML elements
4. For each extracted element E do:
   a. Parse srcML to check if E does exist
   b. If E does not exist then update classML by adding the property 
      \textit{stereotype}="toBeAdded" to the tag of E.

For example, to find methods need to be added to the current code. A simple XPath query is used to get the methods’ names from classML. Then for each method name, srcML is parsed to check if the method does exist or not. Another XPath query is used to look for a method name in srcML.
The search of elements to be added is done in top-down hierarchical way. Classes are checked first. Methods are checked after classes. Finally, relationships are checked after methods.

5.1.3.2 Elements to be Deleted

Finding elements that need to be deleted can be more problematic. To find these elements the process starts with the code. First, the code is checked for the UML elements. Second, the target diagram is checked to determine if elements do exist or not. If the UML element in the source code that is not in the target diagram, then this element must be deleted. The process of identifying elements that need to be deleted from the code based on the target diagram is summarized by the following steps:

1. Represent the target UML diagram in the XML representation classML
2. Represent the current source code in the XML representation srcML
3. Parse srcML to identify all UML elements
   a. Parse classML to check if E does exist
   b. If E does not exist then update classML by adding the property 
      
      stereotype="toBeDeleted" 

      to the tag of E.

   For example to check for generalizations that need to be deleted, srcML is parsed for current generalizations in the code. An XPath query is used to extract the names of classes. Then for each class name, another XPath query is used to extract the superclass(s) for these classes. The extracted generalizations are compared with the generalizations in the classML. Based on this comparison, the property
stereotype=’toBeDeleted’ is added to the corresponding tag for each missing generalization.

Checking for design elements in the code is done in a bottom-up hierarchical way. First, the relationships are checked; second the methods, and finally the classes.

5.1.4 Discussion

There are many potential educational usages for completion diagrams. Ideally, student practice of object-oriented design programming and design involves both individual and collaborative activities. We want the students to individually work and learn how to express their design in UML by implementing the design they have created. First, students separately create and implement a design. Then, as a group, a target design is created. Each student is then given the task of updating their code to that of the target design, using a completion diagram based on the common target design, and their code. This process can help students to mix individual/collaborative work so they better understand how to express their design and their view of the system.

The tool support to render the completion UML diagram is under development. The diagrams shown in the paper were created using MS Visio. We are currently developing a web-based UML class diagram tool. The classes are expressed in classML, and styling is provided using CSS. This will allow us to assess the usefulness of the approach in a controlled experiment. We are also looking into extensions of the technique to visualize a time line of design changes using a series of UML completion diagrams.
5.2 Updating Design Documents

The second proposed method that supports code-design consistency is to automatically update design during code changes. To keep design descriptions (documents) consistent with code, they must be incrementally updated after each code change with design impact. If the implementation of class A has been changed by adding the method M, the UML class diagram must be updated by adding method M to class A. Unfortunately, this process does not usually happen. This is because the process of updating design is mostly done manually and developers usually fail to keep system engineers aware of what they change.

The solution of this problem is to automatically update design documents (or design specifications) after each code change. In this way, design is guaranteed to evolve when code evolves.

5.2.1 The Method

Since srcTracer automatically identifies design changes, it can be easily embedded in another tool to automatically update design documents after each code change (i.e. commit).

The first step is the representation and storing design description or specification. Design specifications can be represented using different formats. One possible representation is the relational database used by [Sutton, Maletic 2007]. Their reverse engineering set of tools (srcTools) save the recovered design in a set of relational database tables.
By using this relational database representation, each design category (type) is represented in one table. There are five tables that correspond to the five design types in UML class diagrams. These tables are CLASSES, METHODS, GENERALIZATIONS, DEPENDENCIES, and ASSOCIATIONS.

For example, GENERALIZATIONS table is used to save generalizations relationships. In this table, there are two main columns (SOURCE and TARGET). So, for the generalization relationship from class A (subclass) to class B (super-class), it will be saved as one tuple where class A is the SOURCE and B is the target. The same format (SOURCE and TARGET) is applied for the other two relationships (associations and dependencies) where SOURCE holds the from class and TARGET holds the to class.

Once the design is in relational database representation, it can be updated by a set of SQL queries. The designTracer (design tracer) framework is proposed to automatically keep track and update design specifications (documents) during code changes. srcTracer is the core of designTracer. The goal of designTracer is to automatically keep design specifications consistent with code during evolution (i.e. preserving code-design traceability).

Figure 5—4 shows the design of designTracer. After each code change (commit) designTracer is triggered to analyze the code change. If the analyzed code change is identified as a design change, then this design change is automatically transformed into SQL query. Finally, the generated queries are executed on the database that holds design specifications. These queries add/delete design elements (tuples) to/from the corresponding database tables that store design specifications.
5.2.2 Updating Design Database

An advantage of srcTracer is that it reports design changes in simple and predefined formats. This makes it easy to automatically transform reported design changes into SQL queries. Each NEW design element reported by srcTracer is automatically transformed into INSERT SQL query, while each reported OLD design element is transformed into DELETE SQL query. Based on the type of the reported design element, the appropriate table is selected to form the query.

Both srcTracer and designTracer were implemented by C++ and the database management system is sqlite3. designTracer automatically generates and executes SQL queries on database tables under the control of sqlite3.

The following are examples of some design changes reported by srcTracer and the corresponding automatically generated SQL queries. Each query is also automatically executed on design database (tables).
Reported design changes by srcTracer:

- **NEW GENERALIZATION FROM** KateLayoutCache **TO**QObject
- **OLD DEPENDENCY FROM** KateLayoutCache **TO**KateEditInfo
- **NEW METHOD** KateLayoutCache::slotEditDone
- **NEW CLASS** KateScriptNewStuff

**Corresponding SQL queries:**

- INSERT INTO generalizations (source, target) VALUES (‘KateLayoutCache’, ‘QObject’)
- DELETE FROM dependencies WHERE source = ‘KateLayoutCache’ AND target = ‘KateEditInfo’
- INSERT INTO methods (class, method) VALUES (‘KateLayoutCache’, ‘slotEditDone’)
- INSERT INTO classes (class) VALUES (‘KateScriptNewStuff’)

In this way, the design database is kept updated and it correctly reflects the design of the current code. At any time, design documents can be generated from the database.

The format of design documents varies. A set of SQL queries can be used to query design specifications database and report the results to the user.

Another design representation could be the XML representation for UML class diagrams classML. Using classML representation, design update is done via a set of XML queries. These queries add/deleted tags from the XML tree that correspond to added/deleted design elements reported by srcTracer.

Regardless of the representation of design, it is necessary and essential to automate the process of generating the queries (database or XML) and modifying the
design specifications. In this way, design specifications (documents) are guaranteed to evolve as the code evolves (i.e. preserving code-design consistency).

5.3 Related Work

The related work is grouped into two main categories; visualization of UML class diagrams and supporting the co-evolution of software artifacts.

5.3.1 UML Visualization

The related work in this area can be grouped into two categories; visualizing differences in UML class diagrams and visualizing architecture and code evolution.

In the field of visualizing differences in UML class diagrams, Seemann and Gudenberg [Seemann, von Gudenberg 1998] presented a graph-based technique to visualize the difference between versions of object-oriented software. Both versions of the source code are reverse engineered into UML class diagrams. These diagrams are compared, merged, and visualized as one UML class diagram with highlighting of the differences. Nguyen [Nguyen 2006] also used a graph comparison structural-oriented differencing approach to detect and visualize differences between software artifacts. Ohst et al. [Ohst, Welle, Kelter 2003a; b] presented an approach to detect and visualize differences between versions of UML (e.g. class diagrams). The difference detection algorithm traverses the AST representations of the UML model. Both versions are merged in one unified document and the differences are highlighted. Treude et al. [Treude, Berlik, Wenzel, Kelter 2007] presented a technique for computing differences between large models.
In the field of visualizing software evolution, Mens et al. presented the *IntensiVE* tool suite. It is based on their intensional view of the source code (meta-logic). This tool supports the co-evolution of code and design. The tool suite verifies and visualizes the conformance and inconsistencies of design with code or other design versions. Our approach differs in the design change identification process and in the idea of the incremental visualizing of design changes. Lanza [Lanza 2001] presented the Evolution Matrix which is used to visualize evolution of software systems. The evolution matrix visualizes classes as boxes, where number of methods is the width and number of instance variables is the height. Wen et al. [Wen, Kirk, Dromey 2007] presented a tool to visualize the evolution history of the architecture and individual components. D’Ambros and Lanza [D’Ambros, Lanza 2008] presented the *Churrasco* framework, which supports software evolution modeling, visualization and analysis through a web interface.

The proposed visualization approach is distinguished from the related work by visualizing (linking) how incremental code changes affect the UML class diagram during evolution. Another distinction is the idea of visualization how far (differences) the current code is from the target design.

### 5.3.2 Artifacts Co-Evolution

In his dissertation, Wuyts [Wuyts 2001] studied the synchronization among software artifacts during evolution. He provided a synchronization framework to build tools that need synchronization between design and implementation. The idea behind the framework is to provide an explicit relation between design and implementation by expressing design as a logic meta program over implementation.
Cazzola et al. [Cazzola, Pini, Ghoneim, Saake 2007] proposed an approach to code evolution (refactoring) that supports the automatic co-evolution of the design models. Their approach relies on a set of predefined metadata that the developer should use to annotate the application code and to highlight the refactoring performed on the code. Then, these meta-data are retrieved through reflection and used to automatically and coherently update the application design models. Actually, the approach is not fully automated since it requires developers to annotate their code changes manually.

In [D'Hondt, Volder, Mens, Wuyts 2000] D’Hondt et al. discussed their approach to manage the synchronization between design and implementation. The approach uses logic programming techniques to express the design as a set of rules or constraints. The approach is called Logic Meta Programming (LMP). They discussed how LMP can be embedded in a development framework to enforce some design rules during implementation.

Mens et al. [Mens, Mens, Wermelinger 2002] proposed the intentional source-code views as a source code modularization mechanism based on crosscutting concerns. The intentional view model is implemented in logic meta-programming language that can reason about and manipulate object-oriented source code directly. Each intentional view corresponds to an important concern. It groups the set of source-code entities that address this concern. In [Mens, Poll, González 2003] two case studies have been presented to show the usefulness in software maintenance and evolution tasks (including checking design consistency). Based on intentional views, the IntensiVE tool suite is presented in [Mens, Kellens 2005]. It allows for the documentation of structural source-
code regularities. The tool suite verifies the conformance of documentation to the implementation and to provide fine-grained feedback when inconsistencies are discovered. In [Mens, Kellens, Pluque, Wuyts 2006] a detailed case study is discussed to show how the model of intentional views and its tools provide support for co-evolving high-level design and source code of a software system.

France and Bieman [France, Bieman 2001] proposed a framework to support managed evolution of OO systems. The framework called multi-view software evolution (MVSE). In MVSE, software evolution is a process in which models are iteratively evaluated and transformed. At the end of each development stage, the products are evaluated and goals for the next stage are refined. The approach is very general and does not directly support code-design co-evolution.

Aldrich et al. [Aldrich, Chambers, Notkin 2002] proposed ArchJava, a small, backwards-compatible extension to Java that integrates software architecture specifications into Java implementation code. ArchJava ensures traceability between architecture and code (mainly communication integrity). It also supports the co-evolution of architecture and implementation. Since ArchJava is a language, it is not lightweight or easy to use. It does not support changes in design models (class diagrams).

Most of the related work in this area is based on logic programming. So, the code and design both are represented in intermediate meta logic representation. The proposed designTracer tool is a lightweight technique without using any logic based or IR methods. The tool also directly supports incremental design changes recovered from small and incremental code changes.
5.4 Summary

Completion diagrams allow for the expression of perfective maintenance tasks, and the monitoring of the completion status. Representing a target design as a UML class diagram allows developers to use a notation they are already familiar with. The developers do not have to specify all changes that must occur to the system, only the resulting design. As developers commit necessary code changes, they get instant feedback as to the status of the design change. For more complicated tasks, e.g., a major API (Application Programming Interface) change, the steps of the change can be expressed as a series of target designs.

Design Tracer (designTracer) helps in keeping design updated and consistent with daily small and incremental code changes. Each code change is analyzed to check if it is a design change. The reported design change is automatically transformed to an update query on design database.
The dissertation addresses four research issues that relate to the consistency of code-to-design traceability links. The goal is to identify, maintain, and preserve code-to-design traceability links during software evolution. The first issue is the automatic identification of breaks in code-to-design traceability from a code change. The second issue is the understanding of how code-to-design traceability change during daily and incremental code changes. The third issue is the identification of critical classes in design based on their involvement in changes to design. The fourth issue is to preserve code-to-design traceability during incremental code changes. This is done by providing a visual aid to guide developers toward target design and by automatically updating design, if necessary, after each code change.

6.1 Concrete Conclusions
The dissertation has presented the following five concrete research contributions:

- The tool srcTracer which automatically identifies breaks in code-to-design traceability from a code change
- A detailed empirical study for the history of design changes to understand how design changes and the benefits from this understanding
- An approach to automatically measure and identify critical classes in the code-design consistency
• A visualization technique (completion diagrams) to guide developers toward correct implementation of code-to-design traceability

• A tool (desTracer) to automatically update design and keep it consistency with code during evolution

6.2 Detailed Conclusions

An approach has been developed that automatically determines if a given source code change impacts the design of the system. The approach uses lightweight analysis and syntactic differencing of the source code. The approach does not require predefined design. It also does not compare the two graphs that represent old and new design models. Design changes are directly recovered from code changes. An efficient tool was developed to support the approach and is applied to an open source system. The evaluation results show that the tool performs better than (error prone) manual inspection.

A detailed empirical study has been presented to understand and investigate how code-to-design traceability links are changed during daily and incremental code changes. The srcTracer tool has been used to automatically identifying commits that impact design of four software systems over three years of history. It was observed that only a small number of changes impact the design and most of these changes are API changes (i.e. methods). It is also shown that most design changes of the commits are small and only few commits have big impact on design. Additionally, it was found that commits with design changes contain more files and significantly more changed lines. The observations also indicate that most bug fixes do not impact the design.
An approach to measure the importance of a class or set of collaborating classes with respect to the design of a system undergoing evolution has been presented. It is found that very few classes are critical to the design in the context of evolution. These classes play a vital role in the evolution of the system design and should be well understood before maintenance of the systems is undertaken. Special attention must be given to these classes to keep the design intact during evolution. The two techniques, itemset mining and hierarchical clustering, were successfully used to automatically identify sets of classes that were impacted by design changes together. This extends the idea of single important classes of the design to sets of classes that when taken as a group, are critical to the design. In general the number of collaborating classes was relatively small, around three to five classes. This appears to agree with standard principles of software design and keeping collaborations to around five or so classes.

A visual aid approach has been proposed to monitor code changes activities. Completion diagrams allow for the expression of perfective maintenance tasks, and the monitoring of the completion status. Representing a target design as a UML class diagram allows developers to use a notation with they are already familiar. The developers do not have to specify all changes that must occur to the system, only the resulting design. As developers commit necessary code changes, they get instant feedback as to the status of the design change.

A tool has been proposed to automatically update design during evolution after each code change if needed. Design Tracer (designTracer) helps in keeping design updated and consistent with daily small and incremental code changes. Each code
change is analyzed to check if it is a design change. The reported design change is automatically transformed to an update query on design database.

### 6.3 Future Work

The approach of identifying design changed is focused to UML class diagram. Our aim is to extend the approach and the tool to detect dynamic design changes, mainly changes in UML sequence diagrams.

In the field of class importance, the future work aims to measure importance of features. We plan to use concept analysis methods to identify features and relate them to their design changes. We also plan to apply the measure on more open source C++ and Java projects. The identification of related classes that work together may be an indication of class-level refactorings or participation in design patterns.

We plan to deeply investigate recovering traceability links among commits based on their textual contents. Hierarchical clustering can be used to cluster commits based on code changes, commit messages, or/and files names.

We plan to extend designTracer tool to a complete framework to maintain code-to-design traceability. The main two features that will be added to designTracer are:

- Measuring the cost of design change to keep developer aware of the change
- Analyzing the history of design changes to recommend good design decisions
APPENDIX A

Detailed Evaluation Results-Collaborative Classes

Table 6-1 Evaluation of itemsets-Kate–Support=2

<table>
<thead>
<tr>
<th>Set Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>689</td>
<td>1361</td>
<td>2416</td>
<td>3691</td>
<td>4612</td>
<td>4626</td>
<td>3676</td>
</tr>
<tr>
<td>Verified</td>
<td>593</td>
<td>903</td>
<td>1166</td>
<td>1284</td>
<td>1109</td>
<td>722</td>
<td>342</td>
</tr>
<tr>
<td>Accuracy</td>
<td>86%</td>
<td>66%</td>
<td>48%</td>
<td>35%</td>
<td>24%</td>
<td>16%</td>
<td>9%</td>
</tr>
<tr>
<td>Set Size</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>Total</td>
</tr>
<tr>
<td>Identified</td>
<td>2278</td>
<td>1078</td>
<td>377</td>
<td>92</td>
<td>14</td>
<td>1</td>
<td>24911</td>
</tr>
<tr>
<td>Verified</td>
<td>111</td>
<td>903</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7135</td>
</tr>
<tr>
<td>Accuracy</td>
<td>5%</td>
<td>66%</td>
<td>1%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>29%</td>
</tr>
</tbody>
</table>

Table 6-2 Evaluation of itemsets-Kate–Support=3

<table>
<thead>
<tr>
<th>Set Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>13</th>
<th>10</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>Verified</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>Accuracy</td>
<td>100%</td>
<td>66%</td>
<td>100%</td>
<td>66%</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>89%</td>
</tr>
</tbody>
</table>

Table 6-3 Evaluation of itemsets-Kate–Support=3

<table>
<thead>
<tr>
<th>Set Size</th>
<th>2</th>
<th>3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>77</td>
<td>16</td>
<td>93</td>
</tr>
<tr>
<td>Verified</td>
<td>77</td>
<td>16</td>
<td>93</td>
</tr>
<tr>
<td>Accuracy</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
### Table 6-4 Evaluation of itemsets-KSpread–Support=4

<table>
<thead>
<tr>
<th>Set Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>443</td>
<td>712</td>
<td>1481</td>
<td>3033</td>
<td>5009</td>
<td>6435</td>
<td>6435</td>
<td>5005</td>
<td></td>
</tr>
<tr>
<td>Verified</td>
<td>353</td>
<td>257(2)</td>
<td>102</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Accuracy</td>
<td>80%</td>
<td>36%</td>
<td>7%</td>
<td>2%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2%</td>
</tr>
</tbody>
</table>

### Table 6-5 Evaluation of itemsets-KSpread–Support=5

<table>
<thead>
<tr>
<th>Set Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>188</td>
<td>116</td>
<td>43</td>
<td>6</td>
<td>353</td>
</tr>
<tr>
<td>Verified</td>
<td>185</td>
<td>115</td>
<td>43</td>
<td>6</td>
<td>349</td>
</tr>
<tr>
<td>Accuracy</td>
<td>98%</td>
<td>99%</td>
<td>100%</td>
<td>100%</td>
<td>99%</td>
</tr>
</tbody>
</table>

### Table 6-6 Evaluation of clustering-Kate

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>13</th>
<th>10</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>Verified</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>Accuracy</td>
<td>100%</td>
<td>66%</td>
<td>100%</td>
<td>66%</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>89%</td>
</tr>
</tbody>
</table>

### Table 6-7 Evaluation of clustering-KSpread

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>15</th>
<th>29</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>32</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>66</td>
</tr>
<tr>
<td>Verified</td>
<td>31</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>56</td>
</tr>
<tr>
<td>Accuracy</td>
<td>97%</td>
<td>80%</td>
<td>86%</td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>85%</td>
</tr>
</tbody>
</table>
APPENDIX B

The Conducted Experimental Study to Evaluate srcTracer

Case 1:
--- ../HippoDraw-1.18.1/reps/AxisRepColor.cxx
+++ ../HippoDraw-1.19.1/reps/AxisRepColor.cxx

```
#include "graphics/DataView.h"
#include "pattern/string_convert.h"
#include "transforms/BinaryTransform.h"
#include "plotters/PlotterBase.h"
#include <algorithm>
#include <functional>

float x = mx + 0.5 * mw;
float y = 30;
if (m_titleFont != 0) {
  y = y + 1.2 * m_titleFont->pointSize() - 11.0; // experimental
}

if (m_zLabelFont != 0) {
  y = y + 1.2 * m_zLabelFont->pointSize() - 8.0; // experimental
  view.drawText (z_label, x, y, 0.0, 0.0, 'c', 'b',
                  false, m_zLabelFont);
} else {
}

drawColorScale (const BinToColor & bin_to_color, ViewBase & base)
{
  DataView & view = dynamic_cast<DataView &> (base);
  PlotterBase * plotter = view.getPlotter();
  const Rect & margin = view.getMarginRect();

  for (float i = 0; i <= margin.getWidth(); i++) {
    Color color;
    const Color & rep_color = plotter->repColor();
    Color color = rep_color;
    bin_to_color.doubleToColor (value, color);
    view.drawViewSquare (margin.getX() + i,
```
Case 2:
--- ../HippoDraw-1.18.1/binners/Bins1DProfile.cxx
+++ ../HippoDraw-1.19.1/binners/Bins1DProfile.cxx

int Bins1DProfile::getNumberOfEntries ( int i ) const
{  
   //   return *( m_num.begin() + i + 1 );
   return *( m_num.begin() + i + 1 );
   
   return static_cast < int > (m_num [ i+1] );
}

Case 3:
--- ../HippoDraw-1.18.1/reps/BinToGreyScale.cxx
+++ ../HippoDraw-1.19.1/reps/BinToGreyScale.cxx

#include "reps/BinToGreyScale.h"

namespace hippodraw {

using namespace hippodraw;

BinToGreyScale::BinToGreyScale ( const char * name )
{  
   doubleToColor ( double value, Color & color ) const
   
   double tmp = 255. - ( value - m_vmin ) / m_dv ;
   double red_tmp = 255. - tmp * ( 255. - static_cast < double > ( red ) );
   double green_tmp = 255. - tmp * ( 255. - static_cast < double > ( green ) );
}
+  double blue_tmp = 255. - tmp * ( 255. - static_cast < double > ( blue ) );
+  int ired_tmp = static_cast < int> ( red_tmp );
+  int igreen_tmp = static_cast < int> ( green_tmp );
+  int iblue_tmp = static_cast < int> ( blue_tmp );
-  color.setColor ( itmp, itmp, itmp );
+  color.setColor ( ired_tmp, igreen_tmp, iblue_tmp );
} // namespace hippodraw

Case 4:
--- ../HippoDraw-1.18.1/reps/ColorBoxPointRep.cxx
+++ ../HippoDraw-1.19.1/reps/ColorBoxPointRep.cxx

#include "graphics/ViewBase.h"
#include "transforms/PeriodicBinaryTransform.h"
+#include "plotters/PlotterBase.h"
#include <cassert>
#include <cmath>
@@ -36,6 +38,7 @@
}{
    BinToColorFactory * factory = BinToColorFactory::instance ();
    m_bin_to_color = factory -> create ( "Rainbow" );
    + m_box_edge = false;
} ColorBoxPointRep::ColorBoxPointRep ( const ColorBoxPointRep & point_rep )
@@ -44,6 +47,7 @@
    BinToColor * btc = point_rep.m_bin_to_color;
    m_bin_to_color = btc -> clone ();
    + m_box_edge = point_rep.m_box_edge;
} ColorBoxPointRep::~ColorBoxPointRep()
@@ -80,6 +84,7 @@
    
    double high = range.high();
    double low = range.low();
@@ -102,7 +107,9 @@
    if ( !user_rect.isInDepth ( value ) ) {
        double x(0), y(0);
color.getRed(),
color.getGreen(),
color.getBlue());
+    if (m_box_edge) {
+        std::vector<double> x;
+        x.push_back(x1);
+        x.push_back(x2);
+        x.push_back(x2);
+        x.push_back(x1);
+        x.push_back(x1);
+        std::vector<double> y;
+        y.push_back(y1);
+        y.push_back(y1);
+        y.push_back(y2);
+        y.push_back(y2);
+        y.push_back(y1);
+        Color color(Color::black);
+        view->drawPolyLine ( x, y, Line::Solid, color, 1.0);
    }
}
+}
+void
+ColorBoxPointRep::setBoxEdge( bool show )
+{ m_box_edge = show;}

Case 5:
--- ../HippoDraw-1.18.1/qt/CreateNTuple.cxx
+++ ../HippoDraw-1.19.1/qt/CreateNTuple.cxx

+#ifdef HAVE_CONFIG_H
+#include "config.h"
+#endif

double max = cutrange.high();
- const std::string & inversion =
-    (cp2->getCutInversionY())?"Inverted":"";
- QtSortedCheckListItem * item = new QtSortedCheckListItem (m_CutListView,dummy);
-     item->setText( 1, QString("%1").arg ( label.c_str() ));
- if ( suffix == ".fits" )
+ if ( suffix == ".fits" || suffix == ".gz" )
+    FitsController * fc = FitsController::instance();
+    // int retVal =
- fc->createNTupleToFile ( m_column_list, m_cut_list, ds, filename, name);
+// fc -> writeNTupleToFile ( m_column_list, m_cut_list, ds, filename, name);
+ fc -> writeNTupleToFile ( ds, filename, name, m_column_list, m_cut_list);
Case 6:

```cpp
void CutController::
  setZoomPan ( PlotterBase * cut_plotter,
  - Axes::Type axis,
  - bool yes )
+setZoomPan ( PlotterBase * cut_plotter, Axes::Type axis, bool yes )
{
  if ( yes )
    m_zoom_pan.push_back ( make_pair( cut_plotter, axis ) );

  return found;
}

+CutController::
+fillAcceptedRows ( std::vector < bool > & acceptArray,
  + const DataSource * source,
  + const std::vector < const TupleCut * > & cut_list )
+
  acceptArray.clear ();
+
  std::size_t size = source->rows ();
  acceptArray.reserve ( size );
+
  std::size_t num_cuts = cut_list.size ();
  for ( unsigned int i = 0; i < size; i++ )
  {
    // If cut is not selected, default is accept.
    bool accept = true;
    +
    // Check all the cuts.
    + for ( unsigned int j = 0; j < num_cuts; j++ )
    +
      const TupleCut * tc = cut_list[j];
      accept = tc->acceptRow ( source, i );
      if (!accept) break;
    +
    + acceptArray.push_back ( accept );
  +}
}
Case 7:
--- ../HippoDraw-1.18.1/datareps/DataRepFactory.cxx
+++ ../HippoDraw-1.19.1/datareps/DataRepFactory.cxx

@@ -30,11 +30,12 @@
#include "St1DHistogram.h"
#include "St2DHistogram.h"
#include "StripChart.h"
+#include "VariableMesh.h"
#include "XYPlot.h"
#include "XYZPlot.h"
#include "YPlot.h"

-namespace hippodraw {
+using namespace hippodraw;

DataRepFactory * DataRepFactory::s_instance = 0;
@@ -56,7 +57,6 @@

void DataRepFactory::initialize () {
@@ -73,9 +73,8 @@
   add ( new St1DHistogram () );
   add ( new St2DHistogram () );
   add ( new StripChart () );
+  add ( new VariableMesh () );
   add ( new XYPlot () );
   add ( new XYZPlot () );
   add ( new YPlot () );
} - // namespace hippodraw

Case 8:
--- ../HippoDraw-1.18.1/graphics/DataView.cxx
+++ ../HippoDraw-1.19.1/graphics/DataView.cxx

#include "DataView.h"
#include "graphics/FontBase.h"
#include "axes/Range.h"
#include "plotters/PlotterBase.h"

    float marginXLeft = draw.getHeight () * 0.20;
    marginXLeft = std::min ( marginXLeft, 55.0f );
-    float marginXRight = 20.0 ;
- float marginYTop = 30.0;
- float marginYBottom = 30.0;

+ // Get a pointer to the plotter.
+ PlotterBase* plotter = getPlotter();
+ 
+ + // Set and adjust top margin
+ float marginYTop = 30.0;
if (m_plotter->hasAxis(Axes::Z))
  {
    marginYTop = 70.0;
  }
+ const FontBase* titlefont = plotter->titleFont();
+ if (titlefont) {
+   marginYTop = marginYTop+titlefont->pointSize()-9.0;
+ }
+ const FontBase* zfont = plotter->labelFont(Axes::Z);
+ if (zfont) {
+   marginYTop = marginYTop+zfont->pointSize()-7.0;
+ }
+
+ + // Set and adjust bottom margin
+ float marginYBottom = 34.0;
+ const FontBase* labelfont = plotter->labelFont(Axes::X);
+ if (labelfont) {
+   marginYBottom = marginYBottom+labelfont->pointSize()-11.0;
+ }

**Case 9:**

```plaintext
--- ../HippoDraw-1.18.1/projectors/DyHist1DProjector.cxx
+++ ../HippoDraw-1.19.1/projectors/DyHist1DProjector.cxx
@@ -192,14 +192,15 @@
     // Get the range.
     const Range & r = m_y_axis->getRange(false);

-   double min = r.low() * m_y_axis->getScaleFactor();
-   double max = r.high() * m_y_axis->getScaleFactor();
+   double scale_factor = m_y_axis->getScaleFactor();
+   double min = r.low() * scale_factor;
+   double max = r.high() * scale_factor;

   const vector< double > & values
       = m_proj_values->getColumn(dp::Y);

   for (unsigned int i = 0; i < values.size(); i++)
     {
-       double val = values[i];
+       double val = values[i] * scale_factor;
         // Add value to sum if its within the range.
         if(val >= min && val <= max)
           sum += val;
```
FigureEditor::
maximumZ ( ) const
{
  /* This is only used to determine if the view is a text view */
  QtView* view = NULL;
  
  #if QT_VERSION < 0x040000
    vector< QCanvasItem* > ::const_iterator first = m_items.begin();
    double max_z = (*first++) -> z ( );
    double max_z = (*first) -> z ( );
    
    /* Get z()-100 for text plotter */
    view = dynamic_cast< QtView*> (*first);
    if (view && view->isTextView())
      max_z = 100;
    
    first++;
    
    while ( first != m_items.end() ) {
      QCanvasItem* item = *first;
    #else
    vector< Q3CanvasItem* > ::const_iterator first = m_items.begin();
    double max_z = (*first++) -> z ( );
    double max_z = (*first) -> z ( );
    
    /* Get z()-100 for text */
    view = dynamic_cast< QtView*> (*first);
    if (view && view->isTextView())
      max_z = 100;
    
    first++;
    
    while ( first != m_items.end() ) {
      Q3CanvasItem* item = *first;
    #endif
    double z = item -> z ( );
    
    /* z()-100 for text */
    view = dynamic_cast< QtView*> (item);
    if (view && view->isTextView())
      z = 100;
    
  
}
+ first++; 
+
+ makeVisible ( item ); 
+ } 
+ } 
+void
+ FigureEditor:: 
+ #if QT_VERSION < 0x040000 
+ makeVisible ( const QCanvasItem * item ) 
+ #else 
+ makeVisible ( const Q3CanvasItem * item ) 
+ #endif 
+ { 
+ if ( item != 0 ) { 
+ addSelectedItem ( item ); 
+
+ double z = maximumZ (); 
+ + /* SetZ for text plotter */ 
+ + const QtView* view = dynamic_cast<const QtView*> (item); 
+ + if (view & & view->isTextView()) 
+ + item->setZ( z + 101.0 ); // ensures it is on top of other text plotters 
+ + else 
+ + #ifdef _MSC_VER 
+ + CanvasEvent * event = new CanvasEvent ( item ); 
+ - qApp -> postEvent ( this, event ); 
+ + #else 
+ + makeVisible ( item ); 
+ + #endif 
+ } 
+ #if QT_VERSION < 0x040000 
+ #ifdef _MSC_VER 
+ CanvasEvent * event = new CanvasEvent ( item ); 
- qApp -> postEvent ( this, event ); 
+ #else 
+ makeVisible ( item ); 
+ #endif 
+ } 
+ if ( e->state() == Qt::ShiftButton ) 
+ + if ( ( e->state() == Qt::ShiftButton ) & & (!getZoomMode()) ) 
+ + else 
+ + if ( e -> modifiers () & Qt::ShiftModifier ) 
+ + if ( ( e -> modifiers () & Qt::ShiftModifier ) & & ( !getZoomMode()) ) 
+ #endif 
+ - sellItem -> setZ ( z + 1.0 ); 
+ + /* SetZ for text plotter */ 
+ + const QtView* view = dynamic_cast<const QtView*> (sellItem); 
+ + if (view & & view->isTextView()) 
+ + sellItem->setZ( z + 101.0 ); // ensures it is on top of other text plotters 
+ + else 
+ + sellItem -> setZ ( z + 1.0 ); // ensures it is on top 
+ + m_rightItem = sellItem; 
+ } 
"}
Case 11:
--- ../HippoDraw-1.18.1/fits/FitsFile.h
+++ ../HippoDraw-1.19.1/fits/FitsFile.h

- int fillFromTableColumn ( std::vector < double > & vec,
+  int fillFromTableColumn ( std::vector < double > & v,
    int column);

- int fillDoubleVectorFromColumn ( std::vector < double > &vec, int column);
+  int fillDoubleVectorFromColumn ( std::vector < double > &vec,
+    int column);

  int fillAxisSizes ( std::vector < long > & vec ) const;

+  void fillShape ( std::vector < int > & shape, int column );

- void writeHDU ( long, int column,
-    std::vector < std::string > names,
-    std::vector < std::string > forms,
-    std::vector < std::string > units,
-    std::string extname );
+  void writeHDU ( long rows, int columns,
+    const std::vector < std::string > & names,
+    const std::vector < std::vector < int > > & shapes,
+    const std::string & extname );

void writeImageHDU ( long x, long y );

- void writeColumn ( int col, long row, std::vector < double > data );

+  void writeColumn ( int c, const std::vector < double > & data );

- void writePix ( long x, long y, std::vector <double > data );
+  void writePix ( long x, long y,
+    const std::vector <double > & data );
Case 12:
--- ../HippoDraw-1.18.1/controllers/FunctionController.h
+++ ../HippoDraw-1.19.1/controllers/FunctionController.h

/** Clears and fills the vector with the top level FunctionRep objects that target @a drep and are contained in @a plotter. A top level FunctionRep object is one that is not a member of a CompositeFunctionRep.
+ objects that target @a data_rep and are contained in @a plotter. A top level FunctionRep object is one that is not a member of a CompositeFunctionRep.
*/
void fillTopLevelFunctionReps ( std::vector < FunctionRep * > & reps,
const PlotterBase * plotter,
- const DataRep * drep ) const;
+ const DataRep * data_rep ) const;

/** Returns the CompositeFunctionRep for which @a rep is a member if it is one, otherwise returns rep. */
const std::string & name );
FunctionRep * addFunction ( PlotterBase * plotter,
- const std::string & name,
FunctionRep * frep,
+ DataRep * drep );
+ DataRep * data_rep );

void saveParameters ( PlotterBase * plotter );

Case 13:
--- ../HippoDraw-1.18.1/datasrcs/NTuple.h
+++ ../HippoDraw-1.19.1/datasrcs/NTuple.h

virtual void addRow ( const std::vector<double> & v );
- int addColumn ( const std::string &,
+ virtual int addColumn ( const std::string &,
const std::vector < double > & column );
- the sizes differ, then throws a DataSourceException object.
+ the sizes differ, then throws a std::runtime_error object.
*/
- void replaceColumn ( unsigned int index,
+ virtual void replaceColumn ( unsigned int index,
const std::vector< double > & data );
- void replaceColumn ( const std::string & label,
- const std::vector < double > & data );
Case 14:
--- ../HippoDraw-1.18.1/plotters/PlotterBase.cxx
+++ ../HippoDraw-1.19.1/plotters/PlotterBase.cxx
+void PlotterBase::setBoxEdge( bool flag ) {
+  +
+  bool
+  +PlotterBase::
+  +getBoxEdge()
+  +{
+    +  return false;
+    +}
+    +
+    +const FontBase *
+    +PlotterBase::
+    +titleFont( ) const
+    +{
+      +  return NULL;
+      +}
+    +
+    +FontBase*
+    +PlotterBase::
+    +labelFont ( hippodraw::Axes::Type axes ) const
+    +{
+      +return NULL;
+      +}
+      +
+      +bool
+      +PlotterBase::isImageConvertable () const
+      +{
+        +  return false;
+        +}
+        +
+        +bool
+        +PlotterBase::isTextPlotter() const
+        +{
+          +  return false;
+          +}

Case 15:
--- ../HippoDraw-1.18.1/projectors/ProfileProjector.cxx
+++ ../HippoDraw-1.19.1/projectors/ProfileProjector.cxx
+#include <stdio.h>
+
ProfileProjector::ProfileProjector( ) :
  BinningProjector ( 1 ),
-  NTupleProjector ( 2 )
+  NTupleProjector ( 3 )
{
  m_binding_options.push_back ( "X" );
  m_binding_options.push_back ( "Y" );
+  m_binding_options.push_back ( "Weight (optional)" );
  m_min_bindings = 2;

BinnerAxisFactory * binner_factory =
  BinnerAxisFactory::instance ();
@@ -87,6 +90,7 @@
  unsigned int cols = m_ntuple->columns () - 1;
  if ( m_columns[0] > cols ) m_columns[0] = cols;
  if ( m_columns[1] > cols ) m_columns[1] = cols;
+  if ( m_columns[2] > cols ) m_columns[2] = cols;

  unsigned int & y_col = m_columns[1];
+ unsigned int & w_col = m_columns[2];
  unsigned int size = m_ntuple->rows();
  + bool have_weight = w_col < UINT_MAX;
  +
  - m_binner->accumulate( x, y);
  + double w = 1.0;
  + if ( have_weight) {
    w = m_ntuple->valueAt ( i, w_col);
  }
  + m_binner->accumulate( x, y, w);
}
}

Case 16:
--- ../HippoDraw-1.18.1/python/PyCanvas.cxx
+++ ../HippoDraw-1.19.1/python/PyCanvas.cxx

#include "controllers/DisplayController.h"
#include "controllers/FunctionController.h"
#include "datasrcs/DataSourceException.h"
#include "datasrcs/NTuple.h"
#include "pattern/FactoryException.h"
#include "plotters/Cut1DPlotter.h"
@@ -33,6 +33,7 @@
#include <stdexcept>
+using std::runtime_error;
using std::string;

using namespace hippodraw;
  m_canvas_proxy = new CanvasViewProxy ( view );
}

PyCanvas::PyCanvas ( )
  : m_canvas (0),
    m_canvas_proxy (0),
    m_has_gui ( false )
{
  + PyApp::lock();
  +
  + m_canvas = new CanvasWindow ();
  + CanvasView * view = m_canvas -> getCanvasView ();
  + m_canvas_proxy = new CanvasViewProxy ( view );
  +
  + PyApp::unlock ();
}

PlotterBase * plotter = display_wrap->display();
+#ifndef _MSC_VER
+    while ( PyApp::hasPendingEvents () ) {} 
+#endif
m_canvas_proxy -> addDisplay ( plotter );
}
else {
@@ -124,6 +138,7 @@
 QtDisplay * PyCanvas::getDisplay ()
 {
     check();
+     PyApp::lock();
     QtDisplay * display = 0;
     PlotterBase * plotter = m_canvas->selectedPlotter();
@@ -210,20 +225,21 @@
 */
 void PyCanvas::saveAsImage ( QtDisplay * display, const std::string &filename )
 {
     - check();
     +// check();
     // Ensure that a suffix is provided...
     std::string::size_type i = filename.find_last_of ( '.' );
     if ( i == std::string::npos ) {
         const std::string
             what ( "PyCanvas::saveAsImage: filename suffix missing." );
         PyApp::unlock ();
     - throw DataSourceException ( what );
     + throw runtime_error ( what );
     }
     QtView * selectedView = findSelectedView ( display );
     if ( selectedView ) {
         std::string file = filename;
         PlotterBase * plotter = selectedView->getPlotter();
         +// while ( PyApp::hasPendingEvents () ) {} 
         m_canvas_proxy -> saveAsImage ( plotter, filename );
     }
     - throw DataSourceException ( what );
     + throw runtime_error ( what );
     }
+// while ( PyApp::hasPendingEvents () ) {} 
+ m_canvas_proxy -> clear ();
}
@@ -414,9 +432,9 @@
     const PlotterBase * plotter = display -> display ();
     QtView * view = m_canvas -> getViewFor ( plotter );
     if ( view != 0 ) {
     - int w = view -> width ();
     - int hh = static_cast < int > ( h );
     - view -> setSize ( w, hh );
+    Rect rect = view -> getDrawRect ();
+    view -> setDrawRect ( rect.getX (), rect.getY (),
+                     rect.getWidth (), h );
Case 17:
--- ../HippoDraw-1.18.1/python/PyFitsController.h
+++ ../HippoDraw-1.19.1/python/PyFitsController.h

```cpp
class FitsController;
class FitsNTuple;
class PyDataSource;
+ class QtCut;

void writeToFile ( const PyDataSource * source,  
          const std::string & filename );

+  void writeToFile ( const DataSource * source,  
+         const std::string & filename,  
+         const std::vector < QtCut * > & cut_list,  
+         const std::vector < std::string > & column_list );
+
+  /** Writes a copy of the PyDataSource @a source to a file, Write to  
+      a file a copy of the DataSource with only columns in the  
+      @a column_list and rows passing all the cuts in the @a cut_list.  
+  */
+  void writeToFile ( const PyDataSource * source,  
+         const std::string & filename,  
+         const std::vector < QtCut * > & cut_list,  
+         const std::vector < std::string > & column_list );
+
```

Case 18:
--- ../HippoDraw-1.18.1/python/QtCut.cxx
+++ ../HippoDraw-1.19.1/python/QtCut.cxx

```cpp
- fc->createNTupleToFile(col
  umn_list, tuple_cut_list, ds, filename, dsname);
+ fc -> writeNTupleToFile(ds, filename, dsname, column_list, tuple_cut_list );
  PyApp::unlock();
  
#else
 +
 +void
 +QtCut::
 +fillCutList ( std::vector < const TupleCut *> & tuple_cut_list,  
+          const std::vector < QtCut * > & cut_list )
+{
+  std::vector < QtCut * >::const_iterator it = cut_list.begin();
+  for ( ; it != cut_list.end(); ++it )
+    {
+      QtCut * cut = *it;
+      cut -> m_plotter-> fillCutList ( tuple_cut_list );
+    }
+}
```
Case 19:

```c++
-#include "pattern/libhippo.h"
+#include "pattern/Observer.h"

#include <map>
#include <string>

-class MDL_HIPPOPLOT_API RootController
+  class MDL_HIPPO
     PLOT_API RootController : private Observer
{

private:
@@
51,6 +51,14 @@
/** The list of opened ROOT files. */
std::map < std::string, TFile * > m_file_map;

+ typedef std::map < const DataSource *, std::string > TupleToFileMap_t;
+ TupleToFileMap_t m_tuple_map;

+ std::vector < std::string > m_ntuple_names;
DataSource * createNTuple ( const std::string & filename,
const std::string & treename );

+ DataSource * initNTuple ( DataSource * source,
+ const std::string & filename,
+ const std::string & treename );
+
+ void fillDimSize ( std::vector < int > & dims,
+ const DataSource * source,
+ const std::string & column );

+ virtual void update ( const Observable * );
+ virtual void willDelete ( const Observable * obs );
```

Case 20:

```c++
+ typedef std::vector < RootBranch * > BranchList_t;
+ std::vector < RootBranch * > m_data;
+ BranchList_t m_data;

+ typedef std::vector < std::vector < double > * > CacheList_t;
- std::vector < std::vector < double > * > m_data_cache;
```
+ CacheList_t m_data_cache;
-  DataSourceException object.
+  std::runtime_error object.

-  double * doubleArrayAt ( unsigned int row, unsigned int column ) const;
+  virtual double * doubleArrayAt ( unsigned int row,
+      unsigned int column ) const;

-  float * floatArrayAt ( unsigned int row, unsigned int column ) const;
+  virtual float * floatArrayAt ( unsigned int row,
+      unsigned int column ) const;

-  int * intArrayAt ( unsigned int row, unsigned int column ) const;
+  virtual int * intArrayAt ( unsigned int row,
+      unsigned int column ) const;

-  unsigned int * uintArrayAt ( unsigned int row, unsigned int column ) const;
+  virtual unsigned int * uintArrayAt ( unsigned int row,
+      unsigned int column ) const;

-  void replaceColumn ( const std::string & label,
-      const std::vector < double > & data );
-  const std::vector < int > & getColumnShape ( const std::string &
-      column ) const;
-  virtual const std::vector < int > & getColumnShape ( unsigned int ) const;
+  virtual void fillShape ( std::vector < int > & v,
+      unsigned int column ) const;

**Case 21:**

--- ../HippoDraw-1.18.1/datareps/VariableMesh.h
+++ ../HippoDraw-1.19.1/datareps/VariableMesh.h

+hippodraw::VariableMesh classes interface.
+Copyright (C) 2006   The Board of Trustees of The Leland
+Stanford Junior University.  All Rights Reserved.
+*/
+
+###_VariableMesh_H_
+
+include "datareps/DataRep.h"
+
+namespace hippodraw { 
+
+/** Displays X Y mesh. 
+*/

![UML diagram](image)
+ @author Paul F. Kunz <Paul_Kunz@slac.stanford.edu>
+ /*
+ *class MDL_HIPPOPlot_API VariableMesh
+ : public hippodraw::DataRep
+ {
+  public:
+  /** The default constructor. */
+  VariableMesh ( );
+  /** The clone function returns an object of its own kind which
+     is a copy of this object at this moment. */
+  virtual DataRep * clone();
+  bool acceptFunction ( int num );
+  /** Returns @c true. Returns @c true PointRep class used by
+     this class are capable of displaying error on the data points.
+     */
+  virtual bool hasErrorDisplay () const;
+  /** Sets the point representation. Sets the point representation to
+     @ point_rep as DataRep::setRepresentation does. Also sets the
+     error display if appropriate.
+     */
+  virtual void setRepresentation ( RepBase * point_rep );
+  /** Sets the axis bindings. Sets binding of the axis with name axis
+     to the DataSource column with label label.
+     */
+  virtual void setAxisBinding ( const std::string & axis,
+                               const std::string & label );
+  /** Sets the axis bindings. Sets binding all axes to the DataSource
+     column with label in the vector. Also set the error display flag
+     in the SymbolPointRep.
+     */
+  virtual void setAxisBindings ( const std::vector < std::string > & bindings );
+  virtual bool hasAxis ( hippodraw::Axes::Type ) const;
+};
+ // namespace hippodraw
++#endif // _VariableMesh
Case 22:
--- ../HippoDraw-1.18.1/qt/WindowController.h
+++ ../HippoDraw-1.19.1/qt/WindowController.h
-WindowController class interface
+hippodraw::WindowController class interface

@@ -37,8 +37,6 @@
  HippoDraw. It can be used as a helper function to an application.

@@ -102,10 +100,15 @@
*/
void resizeCanvas ( CanvasWindow * window );

- /** Positions the Inspector based on size of screen and width of
- Inspector. */
+ /** Positions the Inspector as far right as possible based on size
+ of screen and width of Inspector.
+ */
  void positionInspector ();

+ /** Move the Inspector next to the edge of the Window.
+ */
+ void moveInspector ( CanvasWindow * window );
+
Case 23:
--- ../HippoDraw-1.18.1/plotters/XyPlotter.h
+++ ../HippoDraw-1.19.1/plotters/XyPlotter.h

class AxisRepBase;
class CompositePlotter;
-class FontBase;

  void prepareToDraw ();
+
  bool m_need_update;

protected:

- FontBase* labelFont ( Axes::Type axes ) const;
+ virtual FontBase* labelFont ( Axes::Type axes ) const;

  void setTitleFont( FontBase* font );

- const FontBase * titleFont ( ) const;
+ virtual const FontBase * titleFont ( ) const;

  virtual bool getShowGrid ();
virtual void setBoxEdge(bool flag);
virtual bool getBoxEdge();
virtual void setTransform(TransformBase *);
virtual int getMinEntries();
void setNeedUpdate(bool isChanged);
virtual bool isImageConvertable() const;

Case 24:
--- ../HippoDraw-1.18.1/python/PyDataSource.h
+++ ../HippoDraw-1.19.1/python/PyDataSource.h

-class PyDataSource {
+class PyDataSource
+{
+private:
+  void checkRank ( boost::python::numeric::array array );
public:
  PyDataSource ( const std::string & name, DataSource * source );
  ~PyDataSource();
+ virtual ~PyDataSource();
@@ -145,6 +153,12 @@
  boost::python::numeric::array
  columnAsNumArray( unsigned int index ) const;

+ /** Replaces or add column vector.
+ */
+ void saveColumn ( const std::string & label,
+       const std::vector < double > & v,
+       const std::vector < int > & shape );
+
+ /** Replace or add a column from vector.
+ */
+ void saveColumnFrom ( const std::string & label,
+       const std::vector < double > & array );
+
+ void addRow ( const std::vector < double > & array );
+
+ void append ( const DataSource * source );
+ void append ( const PyDataSource * source );
REFERENCES


International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11, pp. 134-143.


management systems using information retrieval methods", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 4.


2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'09), Vancouver, Canada, pp. 57-64.


