Software Qual J (2011) 19:35–64
DOI 10.1007/s11219-010-9103-x

ICPC 2009

Automatically identifying changes that impact code-to-design traceability during evolution

Maen Hammad · Michael L. Collard · Jonathan I. Maletic

Published online: 29 August 2010

Abstract An approach is presented 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 bringing 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. The results are evaluated and compared against manual inspection by human experts. The tool performs better than (error prone) manual inspection. The developed approach and tool were used to empirically investigate and understand how changes to source code (i.e., commits) break code-to-design traceability during evolution and the benefits from such understanding. Commits are categorized as design impact or no impact. The commits of four open source projects over 3-year time durations are extracted and analyzed. The results of the study show that most of the code changes do not impact the design and these commits have a smaller number of changed files and changed less lines compared to commits with design impact. The results also show that most bug fixes do not impact design.

Keywords Software evolution · Design change · Software traceability · Commit analysis

M. Hammad (✉) · J. I. Maletic
Department of Computer Science, Kent State University, Kent, OH 44242, USA
e-mail: mhammad@cs.kent.edu

J. I. Maletic
e-mail: jmaletic@kent.edu

M. L. Collard
Department of Computer Science, The University of Akron, Akron, OH 44325, USA
e-mail: collard@uakron.edu

1 Introduction

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. 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.

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. Additionally, developers lose the architecture knowledge of the system during evolution (Feilkas et al. 2009). The consistency of the traceability links from the code-to-design is regularly broken during evolution and the design documents soon decay without expensive and time-consuming upkeep. To maintain consistency, each change to the source code must be examined and comprehended to evaluate its impact on the design. 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.

One could 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.

The work presented here 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 directly supports the comprehension of a code change. In addition, by looking at the evolutionary history of a project, we can answer the broader question: *What can we learn from the evolution of design commits (i.e., code changes with design impact)?* This directly supports the comprehension of evolution of code changes and the impact of successive small code changes on design. The broader question is addressed by a detailed empirical study to show the benefits that can be learned.

Our approach, of identifying design changes, does not rely on the existence of explicit traceability links between code and design. Nor do we actually require the existence of a design document (class diagram). That is, all we use 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 it time. The change history of the source code is readily available from *CVS* or *Subversion*.

Each change in the source code is analyzed to see if it meets a set of criteria we developed that categorizes changes as design altering or not (in the context of UML). The analysis and differencing are accomplished in a lightweight and efficient manner using our srcML and srcDiff (Maletic and Collard 2004) tool sets. The results produced indicate if the change impacts the design along with details about the specific design change.

The work presented here extends our previous investigation (Hammad et al. 2009) by applying the developed approach to analyze how commits break code-design traceability during the evolution (i.e., time duration) of a project. Knowing which commits impact the design plays an important role for testing, updating documentation, and project management in general. An empirical study is presented of four open source projects to study the impact of the evolutionary commits on design during 3 years of development.

The paper is structured as follows. First, in Sect. 2, we detail what changes in code results in a design change. This is followed in Sect. 3 by how we automatically identify these design changes. Section 4 presents the validation of the approach. In Sect. 5, a detailed empirical study is presented to investigate how commits impact design during evolution. Threats to validity and limitations are discussed in Sect. 6. The related work is presented in Sect. 7 and followed by our conclusions and future work in Sect. 8.

2 Mapping code change to design change

Here 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. Let us now discuss both types of changes in more detail.

2.1 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. Adding or removing a class has obvious impacts on the class diagram and most likely on the design. Adding a new class can relate to adding new features or extending existing ones. Removing a class may signify a redesign or refactoring of the system.

Likewise, adding or removing a method changes a class's interface and (sometimes) the design. These situations are both relatively easy to identify and map from the code to the class diagram. Figure 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.* A name change (remove and add) also impacts the design document to a degree. Minimally, the class or method should also be renamed in the design document.

```
---  ../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 );
```

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

The addition or deletion of a relationship between two or more classes can drastically affect the design of a system. Generalization is a syntactic issue in C++ and simple to identify. However, association and dependency can be realized in a number of ways in C++ and as such more analysis is necessary to determine if a change occurs to such a relationship.

The code change in Fig. 1 also shows a new class, *QtCut*, being used as a type for a parameter in the method *writeToFile*. This indicates that a new dependency relationship has been established between classes *PyFitsController* and *QtCut*.

Additionally, in Fig. 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 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.

Analysis of the code change in Fig. 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.

2.2 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

```
---   ../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;
```

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

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 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 class diagrams.

Figure 4 has three different examples of code changes between releases 1.18.1 and 1.19.1 of HippoDraw (www.slac.stanford.edu/grp/ek/hippodraw/index.html). In the first

```
---  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();
 }
  } // end namespace
(2, 3, 3)
('M', u'/trunk/KDE/kdelibs/khtml/ecma/kjs_css.cpp', None, None)
Index: kjs_css.cpp
```

Fig. 3 Code changes from two revisions of domparser.cpp (KDE) that do not impact the design

```
- 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)
```

Fig. 4 Three examples of code changes to method signatures that do not impact the design

example Fig. 4 Part A, the function *getRank* is no longer declared *virtual*. The parameter of the function *fillFromTableColumn* has been renamed from ***vec*** to ***v*** in the second example (B). In the third example (C), the macro *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.

3 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 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 et al. 2003), an XML format that supports the (static) analysis required. Second, the code change(s) are represented with additional XML markup in srcDiff (Maletic and 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 5 shows an example of source code and its corresponding srcML representation. Note that each element is marked with its syntactic information.

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 and Collard 2004)

```
// swap two numbers
if( a>b )
{
 t=a;
 a=b;
 b=t;
}
```

(A) Source Code

```
<unit>
<comment type="line">// swap two numbers</comment>
<if>if <condition>( <expr><name>a</name>&gt;<name>b</name>↵
</expr> )</condition><then>
<block>{
 <expr_stmt><expr><name>t</name>=<name>a</name></expr>↵
;</expr_stmt>
 <expr_stmt><expr><name>a</name>=<name>b</name></expr>↵
;</expr_stmt>
 <expr_stmt><expr><name>b</name>=<name>t</name></expr>↵
;</expr_stmt>
}</block></then></if>
</unit>
```

(B) srcML Representation

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

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. 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. Examples of the srcDiff format are given in the following sections as the change identification process is detailed.

3.1 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 Fig. 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 and Collard (2004).

3.2 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 6 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

Fig. 6 The partial srcDiff of ColorBoxPointRep.h. A new method setBoxEdge was added

```
<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>
```

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.

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.

3.3 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 7 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 to identify all new generalizations is:

//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.

```
<diff:common>

<class>class <name>RootController</name>
    <diff:new><super>: private <name>Observer</name></super></diff:new>
<block>{}</block>;</class>

</diff:common>
```

Fig. 7 The partial srcDiff of RootController.h. The supertype Observer forms a new generalization

```
<diff:common>
<function><type>void</type> <name>DataView::prepareMarginRect</name> (
 )
<block>{

<diff:new>
<decl_stmt><decl><type><name>PlotterBase</name>*</type>
<name>plotter</name></decl>;</decl_stmt>
</diff:new>

}</block></function>
</diff:common>
```

Fig. 8 Partial srcDiff of DataView.cxx. The method has a new declaration that uses PlotterBase, producing a potential new dependency

Figure 8 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()]

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:isdeleted()][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 9 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*.

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*.

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:

//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

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

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

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

Fig. 9 Partial srcDiff of FitsNTuple.h. The class exists in both versions. The new data member uses type DataColumn forming a potential new association

(i.e., no usage of class *B* was found), then it is a design change and requires an update of the design document.

4 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 the 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.

4.1 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. HippoDraw was selected because it is an OO system, well documented, and is of medium size. The two releases were selected because they included significant code changes over a large number of files. 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 5,389 new lines added and 1,417 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 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). 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 of which 44 methods were added to new files and the remaining 74 methods were added to existing files.

Table 1 Design changes automatically identified in HippoDraw 1.19.1 by srcTracer tool

Design change		New files	Deleted files	Changed files	Total
Classes	+	6	0	0	6
	−	0	2	0	2
Methods	+	44	0	74	118
	−	0	1	8	9
Generalizations	+	4	0	2	6
	−	0	0	0	0
Dependencies	+	19	0	17	36
	−	0	1	22	23
Associations	+	0	0	4	4
	−	0	0	0	0
Total design changes		73	4	127	204

4.2 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 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. That is, if the results of the tool are indistinguishable from two human experts, then we can conclude that the tool performs equal to an expert. If the two experts consistently answer a question differently from the tool, we can conclude that the tool does not perform as well.

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 the header or the source file were changed

We considered these factors to ensure that we evaluated the tool's performance on all variations of code/design changes, while at the same time only requiring a reasonable amount of time from the experts. If the file selection process was completely random and the number of files that an expert could consider was limited, we may end up with code changes that represent very few variations of design changes (e.g., all files may have only changes in methods), and hence not get complete coverage of the types of design changes. Given these criteria, we selected 24 files from the 175 files for our study (randomly selecting as much 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 experts. Our estimate of the time required to read and answer this number of problems manually was 2 h.

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 h.[1]

Figure 10 shows one of the problems used in 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 Fig. 10). 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 Fig. 10, 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.

5 Results

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

The column *Tool* shows the results of the tool. Columns S_1, S_2, and S_3 represent the results obtained by the three subjects. For example, the tool identified 33 added methods, while each of the three human experts (S_1, S_2, and S_3) 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 (S_1) 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.

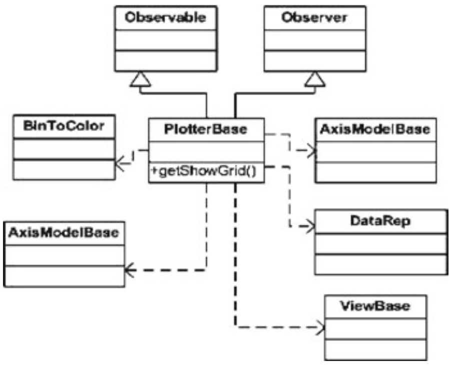
When the intersection is less than what the tool found, there are two possibilities. Either the tool misidentified a change as impacting the design or the expert overlooked a design change. To verify these two possibilities, we need to check the results of the other experts. The second expert also identified 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.

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

[1] Complete study is at www.sdml.info/downloads/designstudy.pdf.

```
---  ../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?

Fig. 10 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

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. For each subject, the design changes identified were a subset of the design changes identified by the tool. The tool identified 47 design changes. The first subject identified 45 design changes (match = 96%), the second subject identified 46 design changes (match = 98%), and the third subject identified 43 design changes (match = 91%). Collectively, the three subjects identified 47 design changes (match = 100%). Therefore, all changes identified by the subjects were also identified by the tool and vice versa.

As such the tool performed *better* than each individual human expert and performed as good as the three subjects together.

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

		Tool	S_1	$S_1 \cap$ tool	S_2	$S_2 \cap$ tool	S_3	$S_3 \cap$ tool	$(S_1 \cup S_2 \cup S_3) \cap$ tool
Classes	+	1	1	1	1	1	1	1	1
	−	0	0	0	0	0	0	0	0
Methods	+	33	32	32	32	32	32	32	33
	−	2	2	2	2	2	0	0	2
Generalizations	+	2	2	2	2	2	2	2	2
	−	0	0	0	0	0	0	0	0
Dependencies	+	7	6	6	7	7	6	6	7
	−	1	1	1	1	1	1	1	1
Associations	+	1	1	1	1	1	1	1	1
	−	0	0	0	0	0	0	0	0

With regards to effort spent, the three subjects required 80 min 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 1 min to run against the entire system.

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.

6 Code-to-design traceability during evolution

The approach and the supporting tool (srcTracer) have been used to conduct a detailed empirical investigation to understand how changes to source code (i.e., commits) impact the design of a software system during the evolution of the system. 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 goal is to systematically understand what percentage of commits actually impacts the design. This automated approach of examining the source code change is very different from investigations of commit messages. Commit messages are often incomplete with regard to design implications and are, in general, problematic to analyze automatically.

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 11 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.

We are interested in investigating how often the design is impacted or changed. 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.

Fig. 11 Part of code changes in revision 723866 on Kate. The change impact impacts design of Kate by adding/deleting three generalizations

```
--- kateprinter.h
+++kateprinter.h
..........
-class KatePrintTextSettings : public KPrintDialogPage
+class KatePrintTextSettings : public QWidget
..........
-class KatePrintHeaderFooter : public KPrintDialogPage
+class KatePrintHeaderFooter : public QWidget
..........
-class KatePrintLayout : public KPrintDialogPage
+class KatePrintLayout : public QWidget
...........
```

6.1 Data collection

The commits of four C++ open source projects over specific time durations are extracted and analyzed. The four projects are the following: 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 history, and vary in their purposes.

The basic unit of data under investigation is the commit and its related information. For each commit, we are interested in the code changes, the date the commit was made, and the number of changed files. For each project, we extracted all commits between two specific dates 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 shows the total number of studied commits (after filtering), total number of added/deleted lines of code in the studied commits, 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 3 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 2,389.

6.2 Categorization of commits

The tool srcTracer and the approach presented in Sect. 3 were used to analyze the code changes of all the extracted commits (Table 3). The code change of each commit was

Table 3 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, the total studied commits (after filtering), and the total investigated code changes

Open source project	Directory	Time period	#Source files	Total commits	Total code changes
Kate	KDE/kdelibs/kate/	3 years (1/1/2006–12/31/2008)	111	1,592	124,772
KSpread	koffice/kspread/	3 years (1/1/2006–12/31/2008)	207	2,389	181,432
QuantLib	trunk/QuantLib/ql/	3 years (1/1/2006–12/31/2008)	671	2,701	782,055
wxWidgets	wxWidgets/trunk/	3 years (1/1/2005–12/31/2007)	3,451	11,438	3,367,630

Table 4 Percentages of commits with design impact and with no design impact for the four projects during the 3-year time period

Project	% Commits-design impact	% Commits-no design impact
Kate	424/1,592 (27%)	1,168/1,592 (73%)
KSpread	681/2,389 (29%)	1,708/2,389 (71%)
QuantLib	924/2,701 (34%)	1,777/2,701 (66%)
wxWidgets	2,269/11,438 (20%)	9,169/11,438 (80%)

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 4. The srcTracer took about 90 min to analyze all the extracted 18,120 commits. During the selected time window, most of the commits to these four projects did not impact the design. Among the 2,701 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 Fig. 12. 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 Fig. 12 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.

From these charts, we can identify periods of time where major design changes occurred and periods of time where design was stable. Design stability can be defined as periods of time where UML class diagram of the system was not (or minimally) changed or impacted by code changes.

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 1,798 new classes added

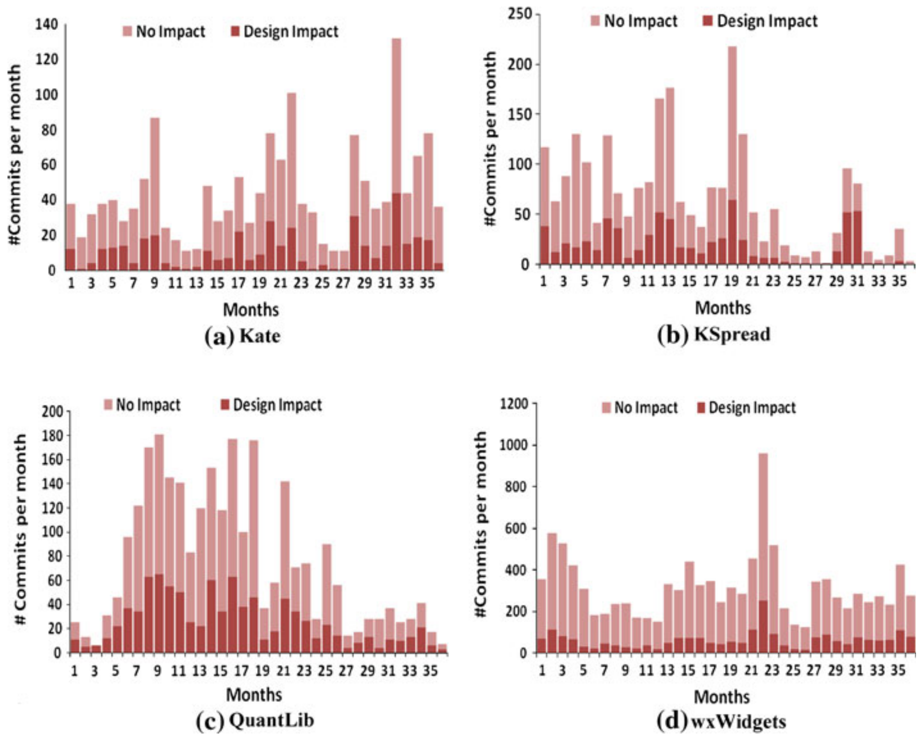


Fig. 12 Histograms of commits to **a** Kate, **b** KSpread, **c** QuantLib and **d** wxWidgets over 3-year time period. For each month, commits are categorized based on impact to design. In almost all months, the number of commits with design changes is much less than the number of commits with no impact to design

during the 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 were 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 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 (Fig. 12b), 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 measures, 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 Tables 5 and 6 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.

Table 5 Distribution of commits based on their design impact category on Kate and KSpread

Design impact	Kate			KSpread		
	+	−	#Commits	+	−	#Commits
Classes	94	70	73	269	198	172
Methods	1,654	727	332	2,673	2,533	525
Generalizations	89	81	73	321	275	174
Dependencies	311	266	177	1,070	697	384
Associations	143	130	104	239	230	157

Table 6 Distribution of commits based on their design impact category on QuantLib and wxWidgets

Design impact	QuantLib			wxWidgets		
	+	−	#Commits	+	−	#Commits
Classes	2,557	422	403	999	310	482
Methods	7,197	1,352	665	8,109	3,691	1,545
Generalizations	1,662	671	329	918	329	430
Dependencies	3,650	766	566	4,289	1,528	1,269
Associations	2,778	769	458	1,422	598	580

The (# Commits) column is the total number of commits for each design type (or design element). For example, during the studied time window, 525 commits impacted the design of KSpread by adding or deleting a method. These 525 commits added 2,673 methods and deleted 2,533 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 impact on the design (as reported by the tool):

```
NEW GENERALIZATION KateLayoutCache 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 Tables 5 and 6 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 1,654. 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 that has the largest number of added classes during the last 3 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 an indicator (or measure) of changes in the degree of coupling/cohesion.

6.3 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 7. The goal of this table is to show that commits with design impact include larger code changes, in terms of both number of changed files and number of changed lines of code, than commits with no design impact.

For the four projects, the average of the number of changed lines by the design impact commits is much 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 lines and files, is shown in Fig. 13. 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% 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.

Also in Kate, 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 9,787 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

Table 7 For each project, the average size of the commit (number of files) and the average number of lines changed (i.e., added and deleted)

Project	Commits			
	Impact on design		No impact on design	
	Avg. #Files	Avg. #Lines	Avg. #Files	Avg. #Lines
Kate	13	217	11	31
KSpread	9	197	10	28
QuantLib	16	510	4	174
wxWidgets	7	1,148	4	83
Avg.	11	518	7	79

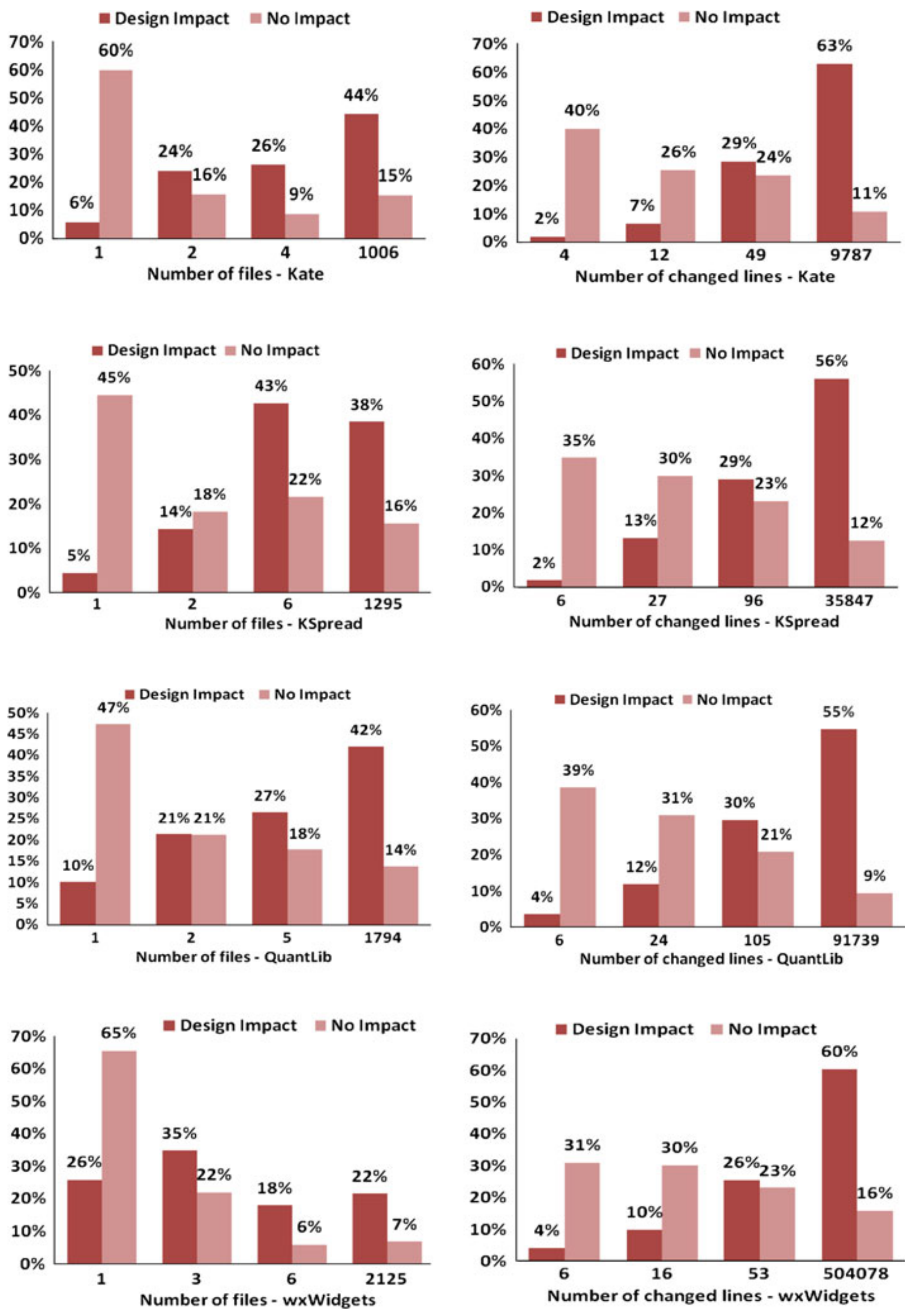


Fig. 13 The distribution of commits into quartiles (x-axis values) based on the number of files and number of lines changed for the four projects. The percentages of commits with design impact and with no impact are shown. In general, commits with a design impact have a larger number of files and consist of more changed lines

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.

6.4 Bug fix versus 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 latter 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 and Votta (2000) and Pan et al. (2009). Using this technique, certain key words in the commit message determine the type of the code change. The approach used in Pan et al. (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.

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 8.

Among the 1,592 commits extracted from Kate, there are 515 (32%) commits categorized as bug fixes. Based on our previous analysis, 424 commits out of the 1,592 are design changes. There are only 97 commits among these 424 (23%) commits that are also 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). 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 et al. 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).

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 by adding the method *KateSearchBar::onCursorPositionChanged*.

Table 8 Grouping of commits based on design impact, bug fixes, and both

	#Bug fix commits	#Design commits	Design ∩ bug fix	Not design ∩ bug fix
Kate	515/1,592 (32%)	424/1,592 (27%)	97/424 (23%)	418/515 (81%)
KSpread	649/2,389 (27%)	681/2,389 (29%)	138/681 (21%)	511/649 (79%)
quantLib	549/2,701 (20%)	924/2,701 (34%)	112/549 (12%)	434/549 (80%)
wxWidgets	3,404/11,438 (30%)	2,269/11,438 (20%)	414/3,404 (18%)	2,990/3,404 (88%)
Avg.	27%	28%	19%	82%

The last column of the table can be used to address the following research question. 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? 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 commits with no design impact. 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.

6.5 Commit labeling

Another application addresses the 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 (with the exact spelling) commit message "*remove wrong signal declaration, obviously i'm STUPID and shouldn't be writing codde at all*". Unfortunately, this commit only tells us what file was changed, and 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 with design changes and hence provide more useful design-level information about the change. Table 9 shows a sample of such labels. The first column is the revision number. The second column represents the type of the design change (Labels). So, in revision 728467 a new dependency relationship was added between classes *KateView* and *KAction*, and this revision is labeled with the appropriate design change.

This labeling can be extracted from an existing revision system from the revision number and the files that were part of the commit.

7 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. The manual construction of the case study may have some possible threats. Design recovery from source code was done by a single author. Some source code elements might be incorrectly reverse engineered. Our selection of the files was not done completely randomly, however, we did assure for a diversity of problems. We could have had an expert check every change but this would have involved a

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

Revision	Labels
513128	OLD CLASS DOMFunction
	OLD METHOD DOMFunction::InternalFunctionImp
728467	NEW DEPENDENCY FROM KateView TO KAction
727168	OLD METHOD KateModeManager::reverse

large amount of effort and we do not feel that any additional information would have been gained.

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 the 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.

For the empirical study, there are some possible threats to the results. One major threat is refactorings. The tool does not deal with all variations for renames (only class rename). The selected durations for the four projects are relatively short and the evolutionary patterns may change for larger durations. Another threat is that we used a lightweight analysis for the commit message to identify bug fixes. There are some bug fixes that cannot be identified using this approach and some developers do not follow the committing guidelines.

8 Related work

Since the paper deals with the traceability issue between code evolution and design changes, the related work is grouped into three categories; software traceability, design changes/evolution, and commit analysis.

8.1 Software traceability

The results of a recent industrial case study (Feilkas et al. 2009) showed that the informal documentation and the source code are not kept consistent with each other during evolution and none of them completely reflects the intended architecture. Developers also are not completely aware of the intended architecture. Antoniol et al. (2000c) 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). In Antoniol et al. (2001), a tool is developed to establish and maintain traceability links between subsequent releases of an object-oriented software system. The method was used to analyze multiple releases of two C++ projects. The focus of the study is the number of added, deleted, and modified LOC, augmented with the LOC of added/deleted classes and methods.

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

IR techniques are used in many approaches to recover traceability links between code and documentation. Antoniol et al. (2000a, 2002) proposed methods based on IR techniques to recover traceability between source code and its corresponding free text documentation. In Antoniol et al. (2000b), 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 (2003) to automatically identify traceability links from system documentation to program source code. De Lucia et al. (2008) used LSI techniques to develop a traceability recovery tool for software artifacts. Zhou and Yu (2007) considered the traceability relationship between software requirement and OO design. 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. (2006) studied and evaluated different requirements tracing methods for verification and validation purposes.

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 and Zisman (2008).

Murphy et al. (2001) presented the software reflexion model technique to summarize a source model of a software system from the viewpoint of a particular high-level model. The technique compares artifacts by summarizing where one artifact is consistent/inconsistent with another artifact. The approach mainly supports the high-level (architectural) model of the system (not UML class diagram). Mappings between a high-level structural model and the source model have to be provided by the user.

8.2 Design evolution

In the area of identifying design changes, Kim et al. (2007) presented an automated approach to infer high-level structural changes of the 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 (2006) presented a technique to identify refactorings. Their identification process is also based on comparing method signatures and the full name of fields and classes. Both of these approaches do not support changes to relationships.

Xing and Stroulia (2005) presented an algorithm (UMLDiff) that 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. UMLDiff was used in Xing and Stroulia (2004a) to study class evolution in OO programs. In Xing and Stroulia (2004b), 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. In Xing and Stroulia (2007), UMLDiff was also used to study and analyze the evolution of API. Another example of graph comparison approaches is presented in Apiwattanapong et al. (2004) and Nguyen (2006). Fluri and 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 and Notkin (2006).

Sefika et al. (1992) proposed a hybrid approach that integrates logic-based static analysis with dynamic visualization for checking design-implementation conformance at various levels of abstraction. The approach does not specifically support UML class diagrams. ArchEvol (Nistor et al. 2005) is proposed as an integrated environment for maintaining and versioning architectural-implementation relationships throughout the development process. Another domain-specific versioning system is Molhado (Nguyen et al. 2005). Molhado supports configuration management for hypermedia and provides version control for individual hyperlinks and document nodes. Aversano et al. (2007) studied evolution of design patterns in multiple releases of three Java projects. The focus of our study is studying incremental design changes from commits not releases.

8.3 Commit analysis

Fluri and 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. (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 work is also done by Hattori and Lanza (2008). Hindle et al. (2008) analyzed and provided a taxonomy for large commits. The goal is to understand what prompts a large commit. They showed that large commits are more perfective while small commits are more corrective. Beyer and Noack (2005) introduced a clustering method, using co-change graphs, to identify clusters of artifacts that are frequently changed together. Xing and Sroulia (2004b) provided a taxonomy of class-evolution profiles. They categorized classes into eight types according to their evolutionary activities.

Purushothaman and 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. (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 et al. (2008) to analyze the influence of evolution activities, mainly refactoring, on software defects. Fluri et al. (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. (2007).

Our approach is distinguished from this related work in multiple ways. The techniques used are lightweight and not IR or logic programming. Only the code changes are analyzed. There is no comparison between the code and a design document/artifact. Unlike most of the others, we discover changes in the relationships, from just code, between classes in the design. The empirical study is distinguished by the categorizing and analyzing commits based on their impact on the design and the benefits that can be inferred from such categorization.

9 Conclusions and future work

Our approach is able to accurately determine design changes based only on the code changes. In comparison with human experts, the tool, based on our approach, performs as well or better than the three subjects. That is, one cannot tell the difference in the quality of the results between human experts and the tool for the studied two releases of the system. Additionally, the tool is very useable with respect to run time while manual inspection is quite time consuming. The tool will greatly reduce the time it takes to determine if a source change impacts the design and thus support continued consistent traceability.

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 our approach, the design differences can be 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. Note that this approach works even if the initial design and source code are not consistent.

A detailed empirical study that used the approach for automatically identifying commits that impact design 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 (methods). Additionally, it was found that commits with design changes contain more files and significantly more changed lines.

The observations also indicate that most bug fix commits do not impact the design (i.e., most bug fix commits do not add/delete classes, methods, or relationships to design).

A related task is that of prioritizing code changes. From the viewpoint of a maintainer, a code change that results in a design change requires closer examination.

The approach can be embedded in a complete reverse engineering suite to maintain design documents during evolution. After each revision or phase, a quick check can be performed on the code changes to determine if the design documents need to be updated (as done by our tool). For any parts of the design impacted, appropriate updates are made to the design documents. Our approach also produces what changes are needed to the design document (however, automatically updating the document is not done here). So, there would be no need to periodically reconstruct or regenerate the design documents. In this way, the consistency of the design documents with code is maintained at less cost.

We are currently working on linking the commits with bug tracking repositories to enhance the accuracy of identifying bug fixing commits. Another direction of the study is to consider developers' contribution on the design. We are investigating the design quality per developer.

Acknowledgments This research is funded in part by the U.S. National Science Foundation under NSF grant CCF 08-11140.

References

Alali, A., Kagdi, H., & Maletic, J. I. (2008). What's a typical commit? A characterization of open source software repositories. In *Proceedings of 16th IEEE international conference on program comprehension (ICPC'08)* (pp. 182–191).

Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2000a). Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of IEEE international conference on software maintenance (ICSM'00), San Jose, CA* (pp. 40–51).

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering, 28*(10), 970–983.

Antoniol, G., Canfora, G., Casazza, G., & Lucia, A. D. (2001). Maintaining traceability links during object-oriented software evolution. *Software-Practice and Experience, 31*(4), 331–355.

Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., & Merlo, E. (2000b). Tracing object-oriented code into functional requirements. In *Proceedings of 8th international workshop on program comprehension (IWPC'00), Limerick Ireland* (pp. 227–230).

Antoniol, G., Caprile, B., Potrich, A., & Tonella, P. (2000c). Design-code traceability for object-oriented systems. *Annals of Software Engineering, 9*(1–4), 35–58.

Antoniol, G., Di Penta, M., & Merlo, E. (2004). An automatic approach to identify class evolution discontinuities. In *Proceedings of 7th international workshop on principles of software evolution (IWPSE'04)*, *Japan* (pp. 31–40).

Apiwattanapong, T., Orso, A., & Harrold, M. J. (2004). A differencing algorithm for object-oriented programs. In *Proceedings of 19th international conference on automated software engineering (ASE'04)* (pp. 2–13).

Aversano, L., Canfora, G., Cerulo, L., Grosso, C. D., & Penta, M. D. (2007). An empirical study on the evolution of design patterns. In *Proceedings of 6th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, Dubrovnik, Croatia* (pp. 385–394).

Beyer, D., & Noack, A. (2005). clustering software artifacts based on frequent common changes. In *Proceedings of 13th international workshop on program comprehension (IWPC'05)* (pp. 259–268).

Collard, M. L., Kagdi, H. H., & Maletic, J. I. (2003). An XML-based lightweight C++ fact extractor. In *Proceedings of 11th IEEE international workshop on program comprehension (IWPC'03), Portland, OR, IEEE-CS* (pp. 134–143).

Cysneiros, G., & Zisman, A. (2008). Traceability and completeness checking for agent-oriented systems. In *Proceedings of 2008 ACM symposium on applied computing, Brazil* (pp. 71–77).

De Lucia, A., Oliveto, R., & Tortora, G. (2008). ADAMS re-trace: A traceability link recovery via latent semantic indexing. In *Proceedings of 30th international conference on software engineering (ICSE'08), Leipzig, Germany* (pp. 839–842).

Feilkas, M., Ratiu, D., & Jurgens, E. (2009). The loss of architectural knowledge during system evolution: An industrial case study. In *Proceedings of 17th IEEE international conference on program comprehension (ICPC'09), Vancouver*, *Canada* (pp. 188–197).

Fluri, B., & Gall, H. (2006). Classifying change types for qualifying change couplings. In *Proceedings of 14th IEEE international conference on program comprehension (ICPC'06), Athens, Greece* (pp. 35–45).

Fluri, B., Wursch, M., & Gall, H. C. (2007). Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of 14th working conference on reverse engineering (WCRE'07)* (pp. 70–79).

Hammad, M., Collard, M. L., & Maletic, J. I. (2009). Automatically identifying changes that impact code-to-design traceability. In *Proceedings of 17th IEEE international conference on program comprehension (ICPC'09), Vancouver, Canada* (pp. 20–29).

Hattori, L. P., & Lanza, M. (2008). On the nature of commits. In *Proceedings of 23rd IEEE/ACM international conference on automated software engineering—workshops (ASE'08)* (pp. 63–71).

Hayes, J. H., Dekhtyar, A., & Sundaram, S. K. (2006). Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering, 32*(1), 4–19.

Hindle, A., German, D. M., & Holt, R. (2008). What do large commits tell us? A taxonomical study of large commits. In *Proceedings of 2008 international working conference on mining software repositories (MSR'08)* (pp. 99–108).

Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution Research and Practice (JSME), 19*(2), 77–131.

Kim, M., & Notkin, D. (2006). Program element matching for multiversion program analyses. In *Proceedings of 2006 international workshop on mining software repositories (MSR'06), Shanghai, China* (pp. 58–64).

Kim, M., Notkin, D., & Grossman, D. (2007). Automatic inference of structural changes for matching across program versions. In *Proceedings of 29th international conference on software engineering (ICSE'07), Minneapolis, MN* (pp. 333–343).

Maletic, J. I., & Collard, M. L. (2004). Supporting source code difference analysis. In *Proceedings of IEEE international conference on software maintenance (ICSM'04)* (pp. 210–219). Chicago, Illinois: IEEE Computer Society Press.

Marcus, A., & Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25th IEEE/ACM international conference on software engineering (ICSE 2003), Portland, OR* (pp. 124–135).

Mockus, A., & Votta, L. G. (2000). Identifying reasons for software changes using historic databases. In *Proceedings of 16th IEEE international conference on software maintenance (ICSM'00)* (p. 120).

Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering, 27*(4), 364–380.

Nguyen, T. N. (2006). A novel structure-oriented difference approach for software artifacts. In *Proceedings of 30th annual international computer software and applications conference (COMPSAC'06)* (pp. 197–204).

Nguyen, T. N., Thao, C., & Munson, E. V. (2005). On product versioning for hypertexts. In *Proceedings of 12th international workshop on software configuration management (SCM'05), Lisbon, Portugal* (pp. 113–132).

Nistor, E. C., Erenkrantz, J. R., Hendrickson, S. A., & Hoek, A. v. d. (2005). ArchEvol: Versioning architectural-implementation relationships. In *Proceedings of 12th international workshop on software configuration management (SCM'05), Lisbon, Portugal* (pp. 99–111).

Pan, K., Kim, S., & James Whitehead, J. E. (2009). Toward an understanding of bug fix patterns. *Empirical Software Engineering, 14*(3), 286–315.

Purushothaman, R., & Perry, D. E. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering, 31*(6), 511–526.

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., & Augustine, V. (2004). Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of 20th IEEE international conference on software maintenance (ICSM'04), Chicago, Illinois* (pp. 188–197).

Ratzinger, J., Sigmund, T., & Gall, H. C. (2008). On the relation of refactoring and software defects. In *Proceedings of 2008 international working conference on mining software repositories* (pp. 35–38).

Ratzinger, J., Sigmund, T., Vorburger, P., & Gall, H. (2007). Mining software evolution to predict refactoring. In *Proceedings of first international symposium on empirical software engineering and measurement (ESEM'07)* (pp. 354–363).

Reiss, S. (2002). Constraining software evolution. In *Proceedings of 18th IEEE international conference on software maintenance (ICSM'02), Montréal, Canada* (pp. 162–171).

Reiss, S. (2005). Incremental maintenance of software artifacts. In *Proceedings of 21st IEEE international conference on software maintenance (ICSM'05), Hungary* (pp. 113–122).

Sefika, M., Sane, A., & Campbell, R. H. (1992). Monitoring compliance of a software system with its high-level design models. In *Proceedings of 18th international conference on software engineering (ICSE'92), Berlin, Germany* (pp. 387–396).

Weißgerber, P., & Diehl, S. (2006). Identifying refactorings from source-code changes. *In Proceedings of 21st IEEE/ACM international conference onautomated software engineering (ASE'06), Japan* (pp. 231–240).

Xing, Z., & Stroulia, E. (2004a). Understanding class evolution in object-oriented software. In *Proceedings of 12th international workshop on program comprehension (ICPC'04), Bari, Italy* (pp. 34–43).

Xing, Z., & Stroulia, E. (2004b). Understanding class evolution in object-oriented software. In *Proceedings of 12th IEEE international workshop on program comprehension (IWPC'04)* (pp. 34–43).

Xing, Z., & Stroulia, E. (2005). UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of 20th IEEE/ACM international conference on automated software engineering (ASE'05), Long Beach, CA, USA* (pp. 54–65).

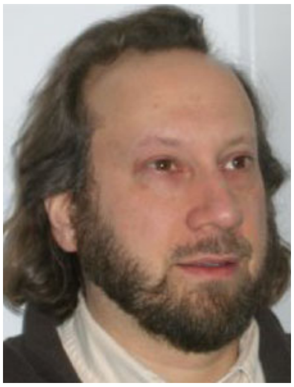
Xing, Z., & Stroulia, E. (2007). API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering, 33*(12), 818–836.

Zhao, W., Zhang, L., Liu, Y., Luo, J., & Sun, J. (2003). Understanding how the requirements are implemented in source code. In *Proceedings of 10th Asia-Pacific software engineering conference (APSEC'03)* (pp. 68–77).

Zhou, X., & Yu, H. (2007). A clustering-based approach for tracing object-oriented design to requirement. In *Proceedings of 10th international conference on fundamental approaches to software engineering (FASE'07), Portugal* (pp. 412–422).

Author Biographies

Maen Hammad completed his Ph.D. at Kent State University in the Department of Computer Science in May 2009 and is now Assistant Professor at Hashemite University, Jordan. He received his Master in computer science from Al-Yarmouk University—Jordan and his B.S. in computer science from the Hashemite University—Jordan. His research interest is Software Engineering with focus on software evolution and program comprehension.



Michael L. Collard is a Visiting Assistant Professor in the Department of Computer Science at The University of Akron in Ohio, USA. His research interests are in source code and source model representation, source code analysis, transformation/refactoring, and differencing for software evolution. He received the Ph.D., M.S., and B.S. in Computer Science from Kent State University.

Jonathan I. Maletic is a Professor in the Department of Computer Science at Kent State University in Ohio, U.S.A. His research interests are centered on software evolution and he has authored over 90 refereed publications in the areas of analysis, transformation, comprehension, traceability, and visualization of software. He received the Ph.D. and M.S. in Computer Science from Wayne State University and the B.S. in Computer Science from The University of Michigan—Flint.