Automatically Identifying Changes that Impact Code-to-Design Traceability

Maen Hammad, Michael L. Collard, and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{mhammad, collard, jmaletic}@cs.kent.edu

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 (i.e., HippoDraw). The results are evaluated and compared against manual inspection by human experts. The tool performs better than (error prone) manual inspection.

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. 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 deign exists. An incremental analysis of the changes, in step with system evolution, should produce a much richer and more actuate 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 also directly supports the comprehension of a code change. Our approach 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 [7] and srcDiff [14] tool sets. The results produced indicate if the change impacts the design along with details about the specific design change.

The paper is structured as follows. First, in section 2 we detail what changes in code results in a design change. This is followed in section 3 of how we automatically identify these design changes. Section 4 presents the validation and threats to validity. In Section 5 is the related work and followed by our conclusions and future work.

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.

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

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.

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 Figure 1 also shows a new class, *QtCut*, being used as a type for a parameter in the method w*riteToFile*. This indicates that a new dependency relationship has been established between classes *PyFitsController* and *QtCut*.

```
--- ../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. Example of code change that impacts design from the addition of a new association relationship between classes FitsNTuple and DataColumn

Additionally, in Figure 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 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.

```
---   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
```

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

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 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 [16]. 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.

```
- 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)
```

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

Figure 4 has three different examples of code changes between releases 1.18.1 and 1.19.1 of HippoDraw[1]. In the first example Figure 4 Part A, the function *getRank* has been declared virtual in the new version. 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

[1] See http://www.slac.stanford.edu/grp/ek/hippodraw/index.html

design. From these results the design document can be updated manually or by some future tool. Output identifying design changes to a file appear 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 [7], an XML format that supports the (static) analysis required. Second, the code change(s) are represented with additional XML markup in srcDiff [14] 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 the XPath, the XML addressing language.

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 [14] 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 they 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 Figure 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 [14].

```
<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 5 The partial srcDiff of ColorBoxPointRep.h. The class exists in both versions, but has a new method setBoxEdge.

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 5 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 XPath srcDiff 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.

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

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

Figure 6. The partial srcDiff of RootController.h. The new supertype Observer forms a new generalization

To identify added/removed generalizations, srcDiff is queried to check any change in the super types of the existing classes. Figure 6 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()]

```
<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>
```

Figure 7. The partial srcDiff of DataView.cxx. The method has a new declaration that uses PlotterBase, producing a potential new dependency.

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 (sub) and B (super), the XPath query is applied to the srcDiff representation of class *A* to identify *B* as an added super type.

Figure 7 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.

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 (or multiplicity) 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']

Table 1. Design changes automatically identified in HippowDraw 1.19.1 by srcTrace tool.

	New Files	Deleted Files	Changed Files	Total
+Classes	6	0	0	6
-Classes	0	2	0	2
+Methods	44	0	74	118
-Methods	0	1	8	9
+Generalizations	4	0	2	6
-Generalizations	0	0	0	0
+Dependencies	19	0	17	36
-Dependencies	0	1	22	23
+Associations	0	0	4	4
-Associations	0	0	0	0
Total Design Changes	73	4	127	204

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.

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 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 HippowDraw

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 approximately 550 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. These lines are distributed over 175 files. There were 160 files, of the 175, that 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). Occurring in new files mean these files do not exist in release 1.18.1, while deleted files means they do 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.

4.2. The Study

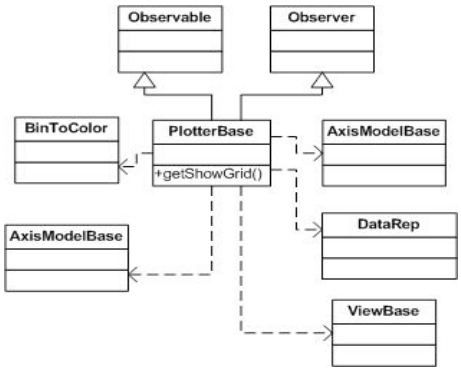
The study compares the results of the tool to that of manual inspection by human experts. Not only will this allow us to determine the accuracy of the tool, we also are able to determine the amount of time spent by experts. This gives us a relative feel for the time savings of using such a tool.

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. For the purposes of this type of

evaluative comparison, a minimum of two human experts is required.

```
--- ../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 8 One test problem given to human experts.

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 only comment changes (for example)
3. File types - we selected situations were only one of header or the source file were changed

Given these criteria we selected 24 files from the 175 files for our study. For each of the 24 sets of changes (one set per file) we develop a problem. This number of problems gave us a diverse set of change types to present to the experts. Additionally, our estimate of the time required to read and answer this number of problems manually seemed to be reasonable for people to accomplish (i.e., less than 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 took approximately 40 hours[2].

Figure 8 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 8). 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 8 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.

4.3. 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 represents 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 newly added methods.

To compare the results of the human experts with the results of the tool, the intersections of the human results with the tool results are shows. For example, there are 33 added methods identified by the tool. The first person (S_1) identified 32 new methods. The intersection of these

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

two sets shows that the tool also identifies the 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 this, we need to check the results of the other experts. The second person 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 person overlooked a different method. The same thing can be said about the 32 methods identified by the third person.

Table 2. Results and comparison 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
-Classes	0	0	0	0	0	0	0	0
+Methods	33	32	32	32	32	32	32	33
-Methods	2	2	2	2	2	0	0	2
+Generalizations	2	2	2	2	2	2	2	2
-Generalizations	0	0	0	0	0	0	0	0
+Dependencies	7	6	6	7	7	6	6	7
-Dependencies	1	1	1	1	1	1	1	1
+Associations	1	1	1	1	1	1	1	1
-Associations	0	0	0	0	0	0	0	0

The rightmost column in the table gives the union of the subjects 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 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 for the other changes, we found that the cumulative results for experts are identical with the results from our tool. All the changes identified by each subject are covered by changes identified by the tool. As such the tool performed *better* than each individual human expert and performed as good as the three subject 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.

4.4. Threats to Validity and Limitations

The study conducted 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 our results here will not scale to other like systems and changes.

Although the 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 maybe very difficult for subject 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 source 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 that this work will map to other languages.

5. Related Work

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

Antoniol et al. [4] 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. [5] proposed an automatic approach, based on IR 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).

IR techniques are used in many approaches to recover traceability links between code and documentation. Antoniol et al. [1, 2] proposed methods based on IR techniques to recover traceability between source code and its corresponding free text documentation. In [3] 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 [15] to automatically identify traceability links from system documentation to program source code. De Lucia et al. [9] used LSI techniques to develop a traceability recovery tool for software artifacts. Zhou and Yu [24] considered the traceability relationship between software requirement and OO design. Zhao et al. [23] 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. [11] studied and evaluated different requirements tracing methods for verification and validation purposes.

Reiss [17, 18] 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. 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 [8].

In the area of identifying design changes, Kim et al. [13] 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 [19] 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.

Xing and Stroulia [22] 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. UMLDiff was used in [20] to study class evolution in OO programs. In [21] 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. Another example of graph comparison approaches is presented in [6]. Fluri and Gall [10] 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. More program element matching techniques are discussed in [12].

Our approach 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, we discover changes in the relationships, from just code, between classes in the design.

6. Conclusions and Future Work

Our approach is able to accurately determine design changes based only on the code changes. In comparison to 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. Additionally, the tool is very useable with respect to run time and manual inspection is quite time consuming. The tool will greatly improve 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 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 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 be 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.

7. References

[1] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "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, October 11-14 2000, pp. 40-51.

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering traceability links between code and documentation", IEEE Transactions on Software Engineering, vol. 28, no. 10, October 2002, pp. 970-983.

[3] Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E., "Tracing Object-Oriented Code into Functional Requirements", in Proceedings of 8th International Workshop on Program Comprehension (ICPC'00), 2000, pp. 227–230.

[4] Antoniol, G., Caprile, B., Potrich, A., and Tonella, P., "Design-code traceability for object-oriented systems", Annals of Software Engineering, vol. 9, no. 1-4, 2000, pp. 35 - 58.

[5] Antoniol, G., Di Penta, M., and Merlo, E., "An automatic approach to identify class evolution discontinuities", in Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE'04), Japan, September 06 - 07 2004, pp. 31 - 40.

[6] Apiwattanapong, T., Orso, A., and Harrold., M. J., "A Differencing Algorithm for Object-Oriented Programs", in Proceedings of 19th International Conference on Automated Software Engineering (ASE'04), 2004, pp. 2-13.

[7] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[8] Cysneiros, G. and Zisman, A., "Traceability and Completeness Checking for Agent-Oriented Systems", in Proceedings of 2008 ACM Symposium on Applied Computing, Brazil, 2008, pp. 71-77.

[9] De Lucia, A., Oliveto, R., and Tortora, G., "ADAMS Re-Trace: A Traceability Link Recovery via Latent Semantic Indexing", in Proceedings of 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 10-18 2008, pp. 839-842.

[10] Fluri, B. and Gall, H., "Classifying Change Types for Qualifying Change Couplings", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens,Greece, June 14-16 2006, pp. 35 - 45.

[11] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K., "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods", IEEE Transactions on Software Engineering, vol. 32, no. 1, January 2006, pp. 4 - 19

[12] Kim, M. and Notkin, D., "Program Element Matching for Multiversion Program Analyses", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR'06), Shanghai, China, 2006, pp. 58 - 64.

[13] Kim, M., Notkin, D., and Grossman, D., "Automatic Inference of Structural Changes for Matching across Program Versions", in Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, May 20 - 26 2007, pp. 333-343.

[14] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11-17 2004, pp. 210-219.

[15] Marcus, A. and Maletic, J. I., "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, May 3-10 2003, pp. 124-135.

[16] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V., "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, September 11 - 14 2004, pp. 188-197.

[17] Reiss, S., "Constraining Software Evolution", in Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02), Montréal, Canada, October 03-06 2002, pp. 162- 171.

[18] Reiss, S., "Incremental Maintenance of Software Artifacts", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Hungary, September 26-29 2005, pp. 113-122.

[19] Weißgerber, P. and Diehl, S., "Identifying Refactorings from Source-Code Changes", in Proceedings of 21st IEEE/ACM International Conference onAutomated Software Engineering (ASE'06), Japan, 2006, pp. 231 - 240.

[20] Xing, Z. and Stroulia, E., "Understanding class evolution in object-oriented software", in Proceedings of 12th International Workshop on Program Comprehension (ICPC'04), Bari, Italy, June 24-26 2004, pp. 34-43.

[21] Xing, Z. and Stroulia, E., "Understanding Phases and Styles of Object-Oriented Systems' Evolution", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, 2004, pp. 242 - 251.

[22] Xing, Z. and Stroulia, E., "UMLDiff: an algorithm for object-oriented design differencing", in Proceedings of 20th IEEE/ACM international Conference on Automated software engineering (ASE'05), Nov. 07-11 2005, pp. 54 - 65.

[23] Zhao, W., Zhang, L., Liu, Y., Luo, J., and Sun, J., "Understanding How the Requirements Are Implemented in Source Code", in Proceedings of 10th Asia-Pacific Software Engineering Conference (APSEC'03), 2003, pp. 68-77.

[24] Zhou, X. and Yu, H., "A Clustering-Based Approach for Tracing Object-Oriented Design to Requirement", in Proceedings of 10th International Conference o Fundamental Approaches to Software Engineering (FASE'07), Portugal, March 24 - April 1 2007, pp. 412-422.