Measuring Class Importance in the Context of Design Evolution

Maen Hammad
Department of Computer Science
Kent State University
Kent, OH 44242
mhammad@cs.kent.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH 44325
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, OH 44242
jmaletic@cs.kent.edu

***Abstract*—A measure of how a class is impacted during design evolution is presented. The history of design changes that involve a given class is the basis for the measure. Classes that are often impacted by design changes are branded as important to the design of the system. Identifying these important classes helps reveal what parts of the system are regularly evolved (e.g., specific features or cross-cutting concerns). The design importance of a class is measured as the number of commits that impact both the design and the class. This is also measured for sets of classes that collaborate to realize a feature or concept in the system. Collaborating classes are identified using itemset mining on commits that impact the design. A small study is presented on two open source projects to illustrate the approach.**

Keywords- impact analysis; software evolution; mining software repositories.

I. INTRODUCTION

As a software system evolves over time its design can drastically change. Classes that once were key players in the design may become less important. Collaborations between classes change to reflect the evolved design and added features. That is, the current list of important classes to understand for a change to be implemented is often very different compared to the list of earlier versions of the system.

Our premise is that classes and collaborations that have been greatly impacted by design changes in recent history are very critical to the current design. Furthermore, these classes and collaborations should most likely be well understood, in the context of the design, before a maintenance task is undertaken.

If this premise holds true, we must first identify which changes impact the design. This will in turn lead us to the classes and collaborations that are most often impacted by these changes.

In the work presented here, we develop a means to measure the importance of classes and collaborations in the context of an evolving design. We leverage our previous work that automatically identifies which changes to a system impact the design [4] and which do not. In short we identify which source code changes add or delete: classes, methods, or relationships between classes. These types of changes impact the design (i.e., UML class diagram) in substantial ways. That is, interfaces are modified, new functionality added or removed, and collaborations between classes changed or added. We label these types of changes as design changes. Our previous work evaluated this approach and found it to be robust at distinguishing between which changes impact the design.

We use this technique to identify design changes and empirically examine the version history of two open source software systems. Our goal is to determine which classes are critical to the evolving design. Furthermore, we automatically identify sets of collaborating classes that are impacted together in the context of design evolution. These sets represent classes, which co-evolve, and participate together to realize a feature or concern.

Our approach does not assume the existence of any design model. All information is derived from the source code and version (commit) history.

The paper is organized as follows. Section II describes the case study. Measuring importance of single classes is detailed in Section III. Identifying collaborative classes and measuring their importance are detailed in Section IV. Related work and conclusions follow.

II. THE STUDY

The evolutionary histories of two C++ open source projects over specific time durations were analyzed. The selected systems are the KDE editor Kate[1], and the KOffice spreadsheet KSpread[2]. These two projects were chosen because they are written in C++, are well documented, have large evolutionary history, and vary in their purposes.

A. *Data Collection*

For each project, a period from its evolutionary history was selected, a target directory was chosen, and all commits between two specific dates from that period were extracted. Start and end dates were selected to cover a duration of three consecutive years. The starting dates were chosen so that both projects were well established and undergoing active development and maintenance. The selected three years periods for the two projects are from 1/1/2006 to 12/31/2008.

[1] See kate-editor.org
[2] See www.koffice.org/kspread

We selected directories that contained the most source and header C++ files. From the set of commits during that time duration those with no C++ code changes are excluded. Test files are not included since they are not part of the design.

TABLE I. INFORMATION ABOUT THE EXTRACTED COMMITS FROM KATE AND KSPREAD

Project	Directory	Time Duration	#Source Files	Total Commits
Kate	KDE/kdelibs/kate	3 years	111	1592
KSpread	koffice/kspread	3 years	207	2389

TABLE II. TOP 5 CLASSES WITH THE HIGHEST IMPORTANCE FOR THE SYSTEMS KATE AND KSPREAD

Kate		Kspread	
Class	**Importance (#Commits)**	**Class**	**Importance (#Commits)**
KateView	81	Sheet	178
KateDocument	72	Cell	148
KateViNormalMode	49	View	106
KateSearchBar	34	Doc	97
KateViewInternal	33	Region	80

TABLE III. DISTRIBUTION OF CLASSES BASED ON THEIR IMPORTANCE FOR KATE AND KSPREAD

Project	Class Importance			
	1	**2-3**	**4-6**	**> 6**
Kate	34% (114/334)	36%	17%	13%
KSpread	28% (154/542)	33%	20%	18%
Avg.	**33%**	**34%**	**18%**	**15%**

Table I presents information about the commits from the projects including the directory in the repository, time period of the duration, the number of C++ header and source files in that directory at the beginning of the duration, and the resulting number of included commits.

B. *Classes Impacted by Design Changes*

Here, only the classes involved in design changes are of interest. Therefore, only the commits that involved design changes are examined. After extracting all commits for a specific time duration, code changes in these commits are analyzed by our tool srcTracer [4] to identify design changes. A *design change* is defined as the addition or deletion of a class, a method, or a relationship (i.e., generalization, association, dependency) in the corresponding UML class diagram. Based on this analysis, commits are categorized as those with design changes (impact) and those with no design changes.

For commits with design changes we extracted the names of all classes that are involved in any design change. For example, the following design changes were reported by srcTracer after analyzing revision 727209 on Kate:

```
NEW METHOD KateLayoutCache::slotEditDone
NEW GENERALIZATION
     FROM KateLayoutCache TO QObject
NEW DEPENDENCY
     FROM KateLayoutCache TO KateEditInfo
```

The names of all classes appeared in these changes (in bold) are extracted. So, revision 727209 will be considered as a design impact that involved classes `KateLayoutCache`, `QObject`, and `KateEditInfo`. Using data mining terminology, the commit is a transaction with three items (i.e., classes).

For Kate, the total number of transactions (commits including design changes) is 424. The total number of distinct items (classes) is 334. For KSpread, there are 681 commits with design changes and 542 unique classes involved in these design changes.

III. IMPORTANCE OF INDIVIDUAL CLASSES

Our approach measures class importance during design evolution based on the number of commits with design changes that impacted the class. The importance of a *single* class **A** is measured as follows:

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

The measure of the importance of a particular class during design evolution is the number of design-change commits involving that class. If most of the commits, with design impact, involved class **A**, then this class is viewed as very important to the design. The impact to a class may be direct, e.g., new methods were added to the class, or indirect, e.g., relationships involving the class were changed.

Based on this measure, Table II presents the 5 most important classes of Kate and KSpread. Classes are ranked according to their importance. The most impacted class by Kate's design changes is the class KateView and for KSpread it is the class Sheet. During the studied three years, 81 revisions committed design changes to Kate involved KateView, either directly or indirectly. These 81 revisions are out of the total number of commits (424) with design impact. So, the importance of KateView during design evolution, compared to all other classes, is 81/424 (19%).

The importance of KateView (ranked 1) shows that it is at the core of many design changes on Kate during the studied three years. It also means that the features related to KateView have undergone considerable evolution. The same applies for the class Sheet in KSpread.

Most classes were involved in very few design changes and only a few classes were impacted by a large number of design changes. The distribution of classes, involved in design changes during the studied three years, based on their importance for the projects is shown in Table III. Classes are grouped into four fixed ranges of importance values (0-1, 2-3, 4-6 and greater than 6). For Kate, 114 classes of all the 334 classes involved in design changes (34%) were impacted by only one commit. For 36% of the classes, their importance values are 2 or 3 commits. The percentage of classes that participated in 4, 5, or 6 commits is 17%. Only 13% of classes have impacted by more than 6 commits. On average, for the two projects 67% of classes were impacted by only 1, 2, or 3 commits which means most classes have low importance values).

```
+class OutputObject:public KJS::JSObject {
+ public:
+ OutputObject(KJS::ExecState *exec,
+          KateDocument *d, KateView *v);
+ virtual ~OutputObject();
+ virtual KJS::UString className()const;
+ void setChangedFlag(bool *flag)
+    { changed = flag; }
+ void setOutputFile
+             (const QString &filename)
+   {this->filename = filename; }
+ private:
+    KateDocument *doc;
+    KateView *view;
+    bool *changed;
+    QString filename;
+    friend class OutputFunction;
+};
```

Figure 1. Example of code change with collaborative classes that impacted together by design changes

IV. Importance of multiple classes

In Section III, we measured the importance of individual classes. Now we extend this measure to the importance of *multiple* classes that collaborate together to implement a feature or concept. Two classes **A** and **B** are collaborative classes if:

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

Fig. 1 shows an example of collaborative classes from the code changes to the file `test_regression.h`. This code change is part of a larger set of changes committed in revision 589090 of Kate. The commit message indicates that this revision included the feature *"added output-object for direct manipulation of result file"*. To implement this feature, revision 589090 impacted the design by adding a new class `OutputObject` with a set of methods. Code change to the class `OutputObject` also indirectly involved the classes `KateView`, `KateDocument` and `KJS::JSObject`. As a result all three classes collaborate to achieve a specific high-level goal that is adding the feature "output an object".

It is important to point out that classes `KateView`, `KateDocument` and `KJS::JSObject` were not changed. The goal is to identify collaborative classes that were involved together in design changes. By including only classes with design changes, we attempt to identify collaborative classes that are mostly responsible for a feature, concept or cross-cutting concern.

The method we used to identify collaborative classes and measure their importance is itemset mining [1]. We represented each commit with design impact as a transaction of classes (items). All classes that impacted the design in the same commit are assumed to be potential collaborative classes. We then apply the Apriori algorithm [1] to all transactions. This well-known algorithm is used to identify co-changed sets of item (*itemsets*). Apriori is a direct technique to identify sets of collaborative classes, with different sizes, and their measure of importance. The method is summarized as follows.

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

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

The distribution of these sets based on size is shown in Table V. For Kate, the size range of the identified itemsets (with support value 3) is from two to six (column Set Size).

The frequency of itemsets with size one (1-itemset) is the measure that we discussed in Section III (importance of individual classes).

TABLE IV. Some identified itemsets (collaborative classes) from Kate by Appriori

Set (Collaborative Classes)	Set Size	Frequency (Importance)
KateDocument KateView	2	24
KateViewInternal KateView KateViNormalMode	3	8
KateCmdActionItem KDialog KateCmdAction KateCmdActionMenu	4	3
Date Character SedReplace CoreCommands SearchCommand	5	3

TABLE V. Results of the identified itemsets by Apriori for Kate (support=3) and KSpread (support=5)

	Kate - Support = 3				
Set Size	2	3	4	5	6
Number of Sets	193	131	52	14	2
	KSpread - Support = 5				
Set Size	2	3	4	5	6
Number of Sets	188	116	43	6	0

V. Related work

Zaidman and Demeyer [13] proposed a technique to identify the most important (key) classes in a system. They defined key classes by classes with a lot of control within the application (controlling functions). Their definition for key classes does not involve the design evolution and differs from our definition. In [11], Xing and Stroulia provided a class evolution taxonomy that consists of eight distinct evolution types. They also employed the Apriori association-rule mining algorithm to discover co-evolving classes. Our works differ in applying Apriori mining to classes that collaborate together in code changes and are not necessarily co-changed. Lanza and Ducasse [6] presented the class blueprint visualization to visualize the internal structure of classes. Sager et al. [10] presented an approach to detect similar Java classes based upon their abstract syntax tree representations. Our work tries to identify collaborative classes that are not necessarily similar. Gîrba et al. [3] proposed the usage of formal concept analysis to identify groups of entities with similar properties that change in the same way and in the same time. For classes, the goal is to identify bad smells. Bieman et al. [2] identified and visualized classes that experience frequent changes together. They call this *change-coupling* between classes. The approach depends on using class-level implementation metrics and class's relationships.

Pinzgert et al. [8] visualized the evolution of source-code entity (e.g. classes and files) and relationship metrics across multiple releases. Robillard [9] proposed a technique based on structural dependencies to automatically discover the program elements (including classes) relevant to a change task. The method depends on static analysis of the source code and does not take into account the history of changes.

Zimmermann et al. [15] Presented a prototype called ROSE that applies data mining methods to version histories in order to guide programmers along related changes. The Apriori Algorithm is used to obtain association rules from version histories. Identifying co-changed lines [14] is an example of applying data mining techniques identify co-changed entities from software repositories. Other example are the identification of call-usage pattern for functions [5] and predicting code changes [12]. Li and Zhou [7] proposed PR-Miner that uses frequent itemset mining to extract general programming rules from large software code. The technique is applied only on the source code of one version.

Our work is distinguished by identifying key classes based on how they impacted the history of design changes. We also identify key sets of collaborating classes based on their involvement in design changes.

VI. CONCLUSIONS & FUTURE WORK

A means to measure the importance of a class or set of collaborating classes in the context of the design of a system undergoing evolution is presented.

We found that very few classes are critical to the design in the context of evolution. That is, only around 15% of all classes in the two systems studied are impacted by more than six design changes. We feel that these classes play a vital role in the evolution of the system design and should be well understood before maintenance of the systems is undertaken. Special attention must be given to these classes to keep the design intact during evolution.

The itemset mining method is used to automatically identify sets of classes that are impacted by design changes together. This extends the idea of single important classes of the design to sets of classes that when taken as a group, are critical to the design.

Our future work aims to verify the accuracy of the identified collaborative classes by the itemset mining technique, and to use clustering as another technique to identify collaborative classes from commits. We also plan to measure the importance of features by using concept analysis methods to identify features and relate them to their design changes. We are working on applying the measure on more open source C++ and Java projects. The identification of related classes that work together may be an indication of class-level refactorings or participation in design patterns will be examined.

REFERENCES

[1] Agrawal, R. and Srikant, R., "Fast Algorithms for Mining Association Rules", in Proceedings of 20th International Conference Very Large Data Bases (VLDB'94), 1994.

[2] Bieman, J. M., Andrews, A. A., and Yang, H. J., "Understanding Change-proneness in OO Software through Visualization", in Proceedings of 11th IEEE Int. Workshop on Program Comprehension (IWPC'03), 2003, pp. 44-53.

[3] Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., and Daniel, R., "Using concept analysis to detect co-change patterns", in Proceedings of 9th International Workshop on Principles of Software Evolution (IWPSE'07), Croatia, 2007, pp. 83-89.

[4] Hammad, M., Collard, M. L., and Maletic, J. I., "Automatically Identifying Changes that Impact Code-to-Design Traceability", in Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada, May 17-19 2009, pp. 20-29.

[5] Kagdi, H., Collard, M. L., and Maletic , J. I., "An Approach to Mining Call-Usage Patterns", in Proceedings of ACM/IEEE International Conference on Automated Software Engineering (ASE'07), 2007, pp. 457-460.

[6] Lanza, M. and Ducasse, S., "A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint", in Proceedings of 16th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, 2001, pp. 300 - 311.

[7] Li, Z. and Zhou, Y., "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code", in Proceedings of 10th European Software Engineering Conference (ESEC-FSE'05), 2005, pp. 306-315.

[8] Pinzger, M., Gall, H., Fischer, M., and Lanza, M., "Visualizing Multiple Evolution Metrics ", in Proceedings of 2005 ACM Symp. on Software Visualization (SoftVis'05), pp. 67-75.

[9] Robillard, M. P., "Automatic generation of suggestions for program investigation", ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, 2005, pp. 11 - 20.

[10] Sager, T., Bernstein, A., Pinzger, M., and Kiefer, C., "Detecting similar Java classes using tree algorithms", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR'06), China, 2006, pp. 65-71.

[11] Xing, Z. and Stroulia, E., "Understanding Class Evolution in Object-Oriented Software", in Proceedings of 12th IEEE International Workshop on Program Comprehension (IWPC'04), 2004, pp. 34 - 43.

[12] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History", IEEE Transactions on Software Engineering, vol. 30, no. 9, 2004, pp. 574-586.

[13] Zaidman, A. and Demeyer, S., "Automatic identification of key classes in a software system using webmining techniques", Journal of Software Maintenance and Evolution: Research and Practice (JSME), vol. 20, no. 6, 2008, pp. 387-417.

[14] Zimmermann, T., Kim, S., Zeller, A., and Jr, E. J. W., "Mining Version Archives for Co-changed Lines", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR'06), 2006, pp. 72-75.

[15] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining version histories to guide software changes", IEEE Transactions on Software Engineering, vol. 31, no. 6, 2005, pp. 429 - 445.