Detecting Breaks in Design Patterns from Code Changes

Maen Hammad
Department of Software Engineering, The Hashemite University, Zarqa, Jordan.
Email: mhammad@hu.edu.jo

Mustafa Hammad
Department of Information Technology, Mu'tah University, AL Karak, Jordan.
Email: hammad@mutah.edu.jo

Ahmed F. Otoom, Mohammad Bsoul
{Department of Software Engineering, Department of Computer Science and Applications}, The Hashemite University, Zarqa, Jordan. Emails:{ aotoom@hu.edu.jo, mbsoul@hu.edu.jo}

***Abstract*—This paper presents an approach to automatically detect and identify breaks in design patterns from a code change during software evolution for C++ programs. The proposed approach aims to determine whether a code change breaks a predefined design pattern or not. The approach analyzes a code change and checks if the change breaks a predefined design pattern that is defined by software designers. Classes and their methods and relationships that are involved in a design pattern are represented in XML format named patternXML with the corresponding design pattern information. After each code change, patternXML file is parsed to determine possible breaks of patterns caused by the committed code change. All identified breaks are saved and archived for future analysis. A simple set of rules are defined to detect and identify breaks in predefined design patterns. The patternXML representation is flexible and can represent different types of design patterns. The approach is realized as a tool and it is evaluated on a set of test cases. Experimental results show that the tool can achieve high accuracy rate in discovering breaks in design patterns from code changes.**

***Index Terms*— software design; design patterns; software evolution**

I. Introduction

As defined by [1], design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. Design patterns are used by designers of software products to solve a specific programming problem. For example, the Composite design pattern is used to group objects in order to treat them in the same way as a single object or instance. Design patterns are widely used and implemented in software systems. Design patterns should be kept intact during software evolution and maintenance activities unless designers decide otherwise.

Design patterns compose of classes, methods and relationships. Each design element of a pattern (i.e. class, method or relationship) plays a specific role in that pattern. Since these elements are code components, they are subject to unauthorized changes by developers during the development process. As a result, patterns are subject to be broken, which affects the quality of the design and the behavior of the system.

The problem is how to enforce design patterns during software evolution. Developers can do a manual check after each code to be sure that they did not break a pattern. Manual checking is a tedious work and consumes times. Furthermore, manual checking is subject to human error. This is because some design patterns are complicated and compose of many design elements which are hard to follow. To identify breaks in design patterns, the whole source code have to be checked periodically. Moreover, the correction of breaks could be very costly. This is because the break has been discovered lately not once it occurs.

Many approaches in the area check the consistency between code and specific design pattern. By using these approaches, consistency is checked by recovering the current pattern from the code and comparing it with the target pattern. In this case, patterns have to be recovered after each code change (i.e. for the new code). Furthermore, a consistency checking is required to be done to compare the recovered pattern with the target pattern. So, a tool is needed to automatically keep track on code changes activities to notify developers once their code change breaks a predefined design patterns.

The research question that is addressed in this paper is*: How to identify breaks in design patterns from a code change?* Designers of software systems define patterns that are suitable for the problem domain. Developers are responsible for implementing these patterns. Furthermore, developers should be aware of keeping these patterns

doi:10.4304/jsw.9.6.1485-1493

intact during maintenance activities. The problem has the following major aspects:

1. How to represent design patterns?
2. How to help designers in defining design patterns?
3. How to analyze code changes to detect breaks in pre-defined design patterns?
4. What are the rules that are used to decide if a code change breaks a pattern or not?

Late discovery of breaks increases the correction cost. On the other hand, failing to discover breaks have direct impact on the behavior of the system since design patterns are designed to solve a specific problem in the problem domain. We try to address these issues by proposing a method and a tool to:

- Automatically detect and identify breaks to reduce manual detecting effort of developers.
- Instantly detect breaks once they occur to reduce the cost of late detection and correction.

In this paper, we present an approach to automatically detect and identify breaks in design patterns once they occur from code changes. The approach does not recover design patterns from source code and compare them with pre-defined patterns. Instead, breaks in design patterns are directly detected from code changes and developers are notified with the cause and the type of the break.

The key in detecting breaks once they occur is the analyzing of design changes caused by code changes. The proposed approach allows designers to define design patterns that they want to preserve during code changes activities. Then, during software evolution, code changes are analyzed to identify design changes. Identified design changes are used to detect breaks in predefined design patterns.

The approach is realized as a tool, named *patternPreserver*, to automatically detect and identify breaks in predefined design patterns from small and incremental code changes. The developed tool keeps track on code changes activities committed by developers to determine breaks. For example, if a developer deletes a class named *Employee* that has a Subject role in the Observer design pattern, patternPreserver notifies the developer who committed the changes as follows:

- WARNING: Break in OBSERVER Pattern.
- DELETE Employee class (SUBJECT).

The warning indicated that the code change caused a break in Observer design pattern. This is because the Employee class has a SUBJECT role in the OBSERVER design pattern. Developers who get this warning should go back and check their code changes carefully. They may have to undo their code changes or get an approval from system's designer to confirm the break. All identified breaks are archived in a database for future analysis. At any time, designers or project managers can analyze historical data about broken design patterns to extract useful information that may help to enhance the design.

The approach is suitable to incremental code changes. The basic unit of change that is considered in this paper is a "commit". After each commit, code changes are analyzed to check possible breaks of predefined design patterns.

This paper is organized as follows. Section 2 details our XML representation for design patterns. Section 3 presents and details the approach. The experimental results of the tool are discussed in Section 4. Threats to validity and limitations of the approach are discussed in Section 5. Related work is presented in Section 6. Section 7 concludes the paper and describes future directions.

II. Representaion of Design Patterns

A design pattern is a design that is implemented in the source code. For example, the Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [1].

Figure 1 shows a simplified UML class diagram for the Observer design pattern. As shown in the figure, the Observer pattern composes of a set of design elements; classes, methods and relationships. Each element has a role in the pattern. For example, the key objects in this pattern are subject and observer; a subject may have any number of dependent observers [1]. All observers are notified once the subject's state changes. Observer class may have one or more subclasses each one plays the Concrete Observer role. Also, the Observer class has a method that has a notification role.

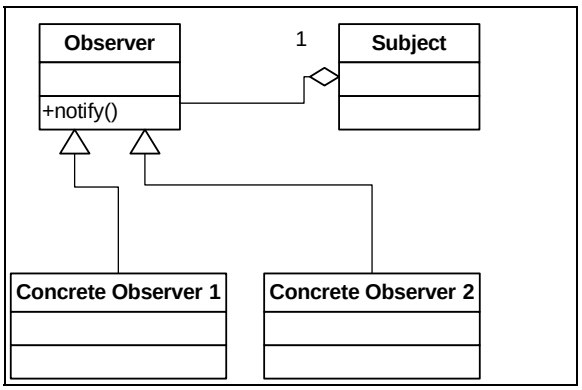

Figure 1: A simplified UML class diagram for the Observer design pattern

Design patterns are usually defined by designers once they build the architecture of the software system. In the created architecture, classes and their design elements are given specific role in specific design patterns. So, design elements and their roles need to be represented and saved in some format. In the proposed approach, design patterns represented and saved in a flexible format. We used XML representation format to save roles of design elements involved in design patterns. This representation is named *patternXML*.

Figure 2 shows a C++ code example of the Observer design pattern. The example shows four classes; Subject, Observer, MinObserver, and MaxObserver. Both MinObserver and MaxObserver are concrete observers. The Subject class defines a collection of observers. The

class has three methods that are used to register, unregister and notify observers. The Subject class has an aggregation relationship with the Observer class. The Observer class has two sub classes (concrete observers) MinObserver and MaxObserver.

```
class Subject {
    vector <Observer*> views;
  public:
    virtual void attach(Observer*);
    virtual void detach(Observer*);
    void notify();
 };

class Observer {
  Subject *model;
 public:
    virtual void update() = 0;
 };

class MinObserver: public Observer {
  public:
    void update();
 };

class MaxObserver: public Observer {
  public:
    void update();
 };
```

Figure 2: A C++ code example for the Observer design pattern.

Figure 3 shows the four classes, shown in Figure 2, represented in our XML format patternXML. patternXML combines basic class information and pattern's information in a single XML file.

Each class is represented with its basic information about the design pattern that it is involved in. The information is shown as XML tags in the patternXML file. For each class, the following information is shown in patternXML file:

(1) the name of the class (<class> <name>)
(2) the pattern which the class involves in (<class> <pattern>)
(3) the role of the class in the pattern (<class> <role>)
(4) the name of the method(s) (<class> <method> <name>)
(5) the role of the method in the pattern (<class> <method> <role>)
(6) the name of the relationship (<class> <relationship> <name>)
(7) the name of the target (outgoing) class involved in the relationship (<class><relationship><to>)
(8) the role of the relationship in the pattern (<class><relationship><role>)

For example, the class MinObserver in Figure 3 (in bold) has the role "concrete observer" in the "observer" design pattern. It has also a method named "update" which plays the role "override base method" in the same design pattern of the class. The class also has a "generalization" relationship to class "observer" with the role "make observer".

The "role" and "pattern" tags in patternXML file are determined by designers. All other tags are filled automatically by the tool. The contents "role" and "pattern" tags are kept empty if their design elements are not involved in any design pattern.

```
<class>
    <name>Subject</name>
     <role>subject </role>
     <pattern> Observer </pattern>
     <method><name>attach</name>
       <role>register observer</role>
     </method>
     <method><name>detach</name>
        <role>unregister observer</role>
     </method>
     <method><name>notify</name>
         <role>notify observers </role>
     </method>
     <relationship><name>Aggregation</name>
         <to>Observer</to>
        <role>observable</role>
     </relationship>
  </class>

<class>
    <name>Observer</name>
    <role>observer</role>
    <pattern>Observer</pattern>
    <method><name>update</name>
      <role>update subject</role>
    </method>
</class>

<class>
   <name>MinObserver</name>
   <role>Concrete observer </role>
   <pattern>Observer</pattern>
   <method> <name>update</name>
       <role>override base method</role>
   </method>
   <relationship>
     <name>Generalization </name>
     <to>Observer</to>
     <role>make observer</role>
     </relationship>
</class>

<class><name> MaxObserver </name>
   <role> Concrete observer  </role>
   <Pattern> Observer </Pattern>
   <method><name>update</name>
     <role>override base method </role>
    </method>
   <relationship>
      <name>Generalization </name>
      <to>Observer</to>
      <role> make observer </role>
  </relationship>
</class>
```

Figure 3: The patternXML representation for the code in Figure 2.

III. The Approach

The approach is summarized in the following steps:

(1) Preserved design patterns are defined by designers and represented in patternXML.
(2) Design changes resulted by commits are identified from code change.
(3) The patternXML file is parsed to check if design changes identified in Step 2 break any defined design pattern.

In the first step, design patterns are defined by designers with the help of patternPreserver tool. Once patterns are defined, patternXML file is generated and saved. These defined patterns should be preserved during code changes activities. Then, after each commit, code changes are analyzed to identify design changes. Finally, patternXML file is parsed based on the identified design changes to check for breaks. The following subsections details these steps.

Our approach is based on the premise that design changes that impact the UML class diagram of the source code is the key to detect breaks in design patterns. So, identifying a design change from a code change leads to identify breaks in predefined design patterns. For example, deleting a method or a relationship, that has a role in a predefined design pattern, results in a warning message to the developer who committed the deletion. In our approach, all code changes that do not affect classes, methods, and relationship are ignored. For example changing a class from an interface to concrete could break a pattern. We do not consider and check this type of code change. The focus is on the design elements of a class.

A. Defining Design Patterns

Designers select appropriate design patterns based on the problem domain. Then, for each pattern, they define classes, methods and relationships to implement the selected pattern. Each defined design element may have a role in that pattern. Determining roles of design elements are done with the help of the patternPreserver tool. patternXML file is automatically generated after the roles of all design elements of a pattern are defined by designers.

All information of the patternXML file, except *role* and *pattern* tags, is generated automatically from the source code of classes. This includes; classes, methods, and relationships. The extracted design elements are shown to the user of the tool. Then, users enter the *Role* and *pattern* information for these extracted elements via the interface of the tool.

The process of extracting design elements (classes, methods, relationships) from the code is done as follows:

1. Source code is represented in the XML representation srcML [2] [3].
2. srcML is parsed by a set of XPath queries:
   a. Classes are extracted.
   b. For each extracted class A:
      i. All its methods are extracted
      ii. Name of super class of A is extracted.
      iii. Names of all non-primitive types that are defined in methods' scopes of class A (dependencies) are extracted.
      iv. Names of all non-primitive types that are defined in the scope of class A (associations) are extracted.

In the first step, source code is transformed into the XML representation srcML. srcML is an XML representation for source code where each code element is tagged with its syntactic information.

Figure 4 shows the srcML representation of class MinObserver shown in Figure 1. As it is shown in the figure, each XML tag represents the syntactic information of each code element. More details about srcML are presented in [2] [3].

In the second step, srcML is parsed by a set of XPath queries that we defined to extract the design elements of the source code. The extracted design elements are classes, methods, and relationships between classes.

XPthat is a query language that is used to extract information from XML files. For example, The XPath query that is used to get classes from srcML is:

//class/name/text()

```
<class>class<name>MinObserver</name>
<super>:<specifier>public</specifier>
   <name>Observer</name></super><block>{
    <private type="default"></private>
    <public> public:
     <function_decl>
     <type><name>void</name></type>
      <name>update</name>
      <parameter_list>()</parameter_list>
      ;</function_decl>
   </public>}</block>;
</class>
```

Figure 4. srcML representation of MinObserver class shown in figure 2.

For each extracted class we check if it has a generalization relationship by using queries that have the following pattern:

//class[name='MinObserver']/super/name/text()

The above query checks and extracts the super class of class "MinObserver". The query extracts the name of the super class "Observer" from srcML in Figure 2. Similar queries are used to extract the methods of a class. All methods' declarations are tagged with "function_decl" in srcML which makes them easy to be extracted.

In order to extract dependencies relationships of a class, we check all local declarations of non-primitive types, which are classes. Local declarations mean any declaration in a method's scope. If the declaration is located in the class's scope as data field, the relationship is considered an association relationship.

All extracted design elements are shown to the user of the tool (i.e. designer) to enable him to determine their roles in design patterns. Classes and their design elements (methods and relationships) are automatically extracted from srcML and shown to designers. After

designers fill design patterns' information for classes under consideration, the patternXML file is generated. The generated patternXML file is used in detecting breaks after each code change.

B. Analyzing Code Changes

The approach is based on direct detection of breaks from code changes. The key is to analyze code changes to identify any addition or deletion to any design element of the system (classes, methods or relationships). Code changed, committed by developers, are analyzed by the *srcTracer* (Source Tracer) [4, 5] tool to automatically identify design changes that affect design. The details about the approach and the tool are presented in [4, 5]. After design changes are identified, patternXML file is parsed to ensure that no predefined design pattern, saved in patternXML, has been broken by committed code changes.

C. Parsing patternXML

patternXML file is parsed by using a set of XPath queries that we defined. The file is parsed if the code change is a design change. Minor code changes that do not impact design do not break a pattern. For example, adding a condition, changing a type for a parameter, or updating a loop have no impact on patterns. But, if a code change adds or deletes a design element, then *role* and *pattern* tags are checked for that design element in patternXML file. For example, if a code change resulted in deleting method Class1::M1, patternXML file is parsed to check if M1 has a pre-defined role in a design pattern (*role* tag). The XPath query is as follow:

//class/method[name="M1"]/role/text()

The above query extracts the role of method Class1::M1 from patternXML representation of class Class1. To extract the name of the design pattern of class A, the following query is used:

//class/method[name="M1"]/pattern/text()

The extracted contents of these tags are used to determine whether deleting method M1 breaks a design pattern. In this case, the following warning message is shown to the committer who deleted method Class1.M1:

- WARNING: Break in PATTERN Pattern.
- DELETE Class1.M1 method (ROLE).

The PATTERN and ROLE keywords are replaced by the information extracted by the two XPath queries mentioned above.

The process of identifying breaks in design patterns from code changes is detailed as follows:

(1) After each commit, design changes are identified by srcTracer.
(2) If the design change is a deletion of a class:
  (a) Parse patternXML to check if the class has a role or one of its methods has a role.
(3) If the design change is a deletion of a method Class1.M1:
  (a) Parse patternXML of class Class1 to check if its method M1 has a role.
(4) If the design change is a deletion of a relationship (generalization, dependency, or association) from class A to class B:
  (a) Parse patternXML of class A to check if the relationship has no role.
(5) If the design change is an addition of a relationship (generalization, dependency or association) from class A to class B:
  (a) Parse patternXML of classes A and B to check if any class has a pattern.

In step one, design changes are identified as discussed in section 4.2. Based on the type of identified design changes one or more of the 2-5 Steps are applied. In case the design change is the deletion of a class, Step two verifies if the deleted class has no role. This is done by parsing the role tag of the deleted class in the patternXML file. The same is applied in Step three for the deleted methods.

Step four is applied if a design change resulted in deleting a relationship from class A to class B. The patternXML representation of class A is parsed to check if the deleted relationship (*name* and *to* tags) does not have a role (*role* tag) in the design pattern (*pattern* tag). patternXML file of class B is also parsed to extract the role of class B (if it has a role). For example, in Figure 1, if a design change resulted in deleting the generalization relationship between class MinObserver and Observer, the following warning is shown to the user:

- WARNING: Break in OBSERVER Pattern.
- DELETE Generalization from MinObserver (CONCRETE OBSERVER) to Observer (OBSERVER).

If the identified design change is an addition of a relationship from class A to class B (Step five), then both pattern and role tags are parsed for classes A and B. Some new relationships may weaken or break the pattern. For example, in observer design pattern, it is a poor design decision to make the subject class inherits from the concrete observer class. It is the designer' decision to determine whether an added relationship break a design pattern. Thus, if any of the two classes which are involved in the new relationship has a pattern, a notification is shown to the developer as a warning for a possible break in design pattern. For instance, if the code change shown in Figure 2 adds a generalization relationship from the concrete observer class MinObserver to the subject class Subject, the following warning is shown to the developer:

- WARNING: Possible Break in OBSERVER Pattern.
- NEW Generalization from MinObserver (CONCRETE OBSERVER) to class (SUBJECT).

The word "Possible" is shown because the design change adds a new relationship (generalization).

a. Archiving and Analyzing Breaks

The patternPreserver tool automatically stores all identified breaks in an XML format that can be used as an archive for violations. Each identified break in a design pattern is saved with its related information. For example, the following identified break is saved in the XML format shown figure 5.

- WARNING: Break in OBSERVER Pattern.
- DELETE Employee class (SUBJECT).

```
<break>
    <pattern> observer  </pattern>
    <developer> malc </developer>
    <date>  01/04/2012 </date>
    <change>
        <action> delete </action>
        <element> class </element>
        <name> Employee </name>
    </change>
</break>
```

Figure 5. A XML representation for a break in Observer design pattern caused by deleting a class.

Each <break> </break> tag represents a break in a design pattern committed by a developer. The broken pattern, developer's ID, and the date of the change are saved in <pattern>, <developer> and <date> tags respectively. The design change that caused the break is also saved in the <change> tag. The type of change (i.e. add or delete), is tagged by the <action> tag. The added/deleted design element and its name are also saved.

The patternPreserver tool parses the archive of violations to provide the following information to the developers or project managers:

1. The most vulnerable design pattern.
2. The developer(s) that has the largest number of breaks
3. The most frequent change that break patterns.

The first information reports the pattern with the large number of breaks. Identifying that pattern may lead designers to choose a different design pattern for the design. This is because the current pattern is not flexible enough to support large number of maintenance tasks or it does not fit properly to the problem under consideration. The second information helps designers to identify developers with poor design knowledge. The third information may help designers to monitor or restrict specific design changes. For example, if most changes that break patterns are resulted from deleting classes, team leaders may prevent developers from deleting a class before getting an approval from designers.

IV. Experimental Results

The evaluation focuses on the correctness and the efficiency of the tool in detecting breaks for predefined design patterns from code changes. We need to test if the approach and the tool can correctly detect violations. A set of test cases were designed for testing. These test cases are set of maintenance tasks that are implemented by selected human subjects. Results obtained from the test cases were manually checked. The manual checking covers all code change committed by human subjects.

Each test case composes of a set of classes that do a specific task. Each test case is an instance of a specific design pattern with a set of maintenance tasks. The number of designed test cases is five. Each test case covers one of the following five design patterns; Adapter, Composite, Factory, Strategy and Observer. So, each design pattern is covered by one test case.

All source code of the five test cases is taken, with some changes, from the code examples provided by an online tutorial for design patterns [6]. Actually, there is a lack in well documented open source project written in C++ that shows exactly how design patterns are implemented in that project. Dong et al. [7] presented a review on current techniques for discovering architecture and design patterns from object-oriented systems. They noticed that the experimental systems, especially the open-source systems, do not provide any architecture and design documents that clearly identify patterns and their locations. Therefore, we used code examples that implement design patterns for the evaluation of the tool.

We carefully designed the programming tasks for each test case to cover most of the possible violations in design patterns. Most of the programming tasks focus on design changes. Deleting of a method, relationship, or a class are examples for programming tasks with design changes. Changing the data type of a variable is an example for a programming task with minor code change.

For each test case, subjects are asked to implement the programming (maintenance) tasks, and then commit their code changes. After each commit, the tool analyzes the change and saves its results. After completing all the maintenance tasks of the five test cases, tool's results were checked manually to determine the correctness of the results.

After completing each maintenance task, the tool's results will be either a warning message about a break or null. A manual checking is performed to determine wither the result is correct or not. This includes the following cases:

- The code change breaks a pattern and the tool correctly reported the break as a warning.
- The code change breaks a pattern and the tool failed to report the break as a warning (**false negative)**.
- The code change does not break a pattern and the tool correctly does not report a warning.
- The code change does not break a pattern and the tool reported a warning **(false positive).**

For example, changing a data type of a variable does not violate the pattern. So, the tool's result is correct if the tool does not report a warning. On the other hand, the result is incorrect if the tool reported a warning message.

The total number of programming tasks for the five test cases is 15, three tasks for each test case. Three

human subjects were asked to test the tool. These subjects are graduate and undergraduate software engineering students who have knowledge in design patterns and C++ programming. A summary tutorial about design patterns were given to subjects. We used more than one subject to cover different programming styles by developers. A summary of the results are shown in Table 1.

Table 1 shows that most of the tool's results are correct for the three subjects. Results were correct for 38 out of the 45 tasks. Five false positive results were reported. Two subjects renamed three methods and two classes. As a result, the tool reported these renames as breaks in design patterns, but they are not. In our approach, we do not handle renaming. If a method M1 has been renamed to M2, the tool considers it as a deletion for method M1. All reported false positive cases are because of renaming issue. No false negative results were reported. This indicates that the tool has a very good performance in identifying breaks in design patterns from code changes.

TABLE 1.
EVALUATION OF THE TOOL'S RESULTS

	Correct	Incorrect (false negative)	Incorrect (false positive)
Subject 1	13/15	0/15	2/15
Subject 2	12/15	0/15	3/15
Subject 3	15/15	0/15	0/15
Accuracy	38/45 (89%)	0/15	5/45 (11%)

V. THREATS TO VALIDITY & LIMITATIONS

The case study that was used in the evaluation, it has three issues that may weaken the results. The fist issue is the number of tested patterns. The tool has been only tested on five design patterns. The second issue is the design of the programming tasks. They may have some limitations in covering all possible design changes that affect patterns. The third issue is the size of test cases. The source code of test cases is small. We did not use official, such as open source project, or large source code. This is because it is hard to find open source C++ project that provides a code with a documentation that clearly define the design pattern for that code.

The approach focuses on the UML design elements as the key to identify breaks. Some minor elements, as interfaces, that may break patterns are not considered. The tool does not consider renames in the process of identifying breaks. For example, if a method is renamed, it will be considered as deleted.

VI. RELATED WORK

Antoniol et al. [8] presented an approach to trace OO design to implementation. The goal was to check the compliance of OO design with source code but not design patterns. Lovatt et al. [9] presented a conventional compiler but extended to include the extra checks needed to enforce design patterns. Patterns are enforced at the class level; the class has to be written to conform to the pattern. The class should implements a predefined interface for the specified pattern. Our proposed approach is for C++ not java and we use a different method to represent design patterns.

Blewitt et al. [10] presented a pattern specification Prolog-like language called SPINE, to allow patterns to be defined in terms of constraints on their implementation in Java. We used a simpler XML representation for patterns. Eichberg et al. [11] presented an approach to express constraints on structural dependencies between program elements to avoid erosion of the intended structure of the code. The approach defines a new logic-based language called LogEn to express ensembles and constraints on their dependencies.

Zhao et al. [12, 13] proposed an approach for design pattern evolution and verification using graph transformation. The transformation is done based on predefined graph transformation rules for each type of pattern evolution. Kim and Shen [14] proposed an approach to evaluating the conformance of class diagrams described in UML to pattern specifications described Role-Based Meta-modeling Language. Zhu et al. [15] presented a tool called LAMBDES-DP to support the use of design patterns during development. Its theoretical foundation is a descriptive semantics of UML in first order logic, and the design patterns are formally specified in the same language.

Balanyi and Ferenc [16] developed a method to discovering design patterns in the source code. It uses specifications of how the patterns work by describing basic structural information like inheritance, composition, aggregation and association. They also presented a XML–based language, called (DPML) Design Pattern Markup Language to represents design patterns. Bayley and Zhu [17] proposed a formal meta-modeling approach that uses first-order predicate logic to specify structural and behavioral features of design patterns. Another design pattern recovery approaches are presented in [18], [19], [20], and [21]. Dong et al. [22] presented an approach to visualize design patterns. They presented a UML profile that defines new stereotypes, tagged values, and constraints for tracing design patterns in UML diagrams. These new elements are attached to a modeling element to explicitly represent the role the modeling element plays in a design pattern. In our approach we focus on preserving design patterns during code changes activities.

Jain and Yang [23] analyzed a medium size commercial OO C++ system to investigate the relationship between patterns, design attributes, and the number of changes. They found that classes that participate in design patterns are not less change prone and these pattern classes are among the most change prone in the system.

Our approach differs from related work in the field in focusing on identifying breaks in design patterns from incremental C++ code changes. It also presents a flexible and easy to extend XML format to represent design

patterns.

VII. Conclusions and Future Work

An approach is presented to automatically identify breaks of design patterns during software evolution. It enables designers to enforce their design patterns during maintenance activities committed by developers. A flexible XML format is also presented to represent design patterns.

The proposed approach is realized as a tool to automatically keep track on code changes and to notify developers with breaks of patterns. The tool keeps tracks on identified breaks by storing them in a historical database. The database is analyzed to extract useful for designers. The tool has been evaluated and its reported accuracy is 89% with 11% false positive rate. The tool can be implemented as a plug-in tool in an IDE, as Eclipse, to support development.

Our future work aims to develop a tool to recover design patterns from C++ source code. This future tool will be combined with the tool developed in this work to build a complete framework for design patterns recovery and enforcement. We also are considering developing another version of the tool for Java source code.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[2] M.L. Collard, J.I. Maletic and A. Marcus, "Supporting Document and Data Views of Source Code," *Proc. 2nd ACM Symposium on Document Engineering (DocEng'02)*, 2002, pp. 34-41.

[3] M.L. Collard, H.H. Kagdi and J.I. Maletic, "An XMLBased Lightweight C++ Fact Extractor," *Proc. 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, 2003, pp. 134-143.

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

[5] M. Hammad, M.L. Collard and J.I. Maletic, "Automatically Identifying Changes that Impact Code-to-Design Traceability during Evolution," *Software Quality Journal*, vol. 19, no. 1, 2011, pp. 35-64.

[6] SourceMaking, 2012; http://sourcemaking.com/design_patterns.

[7] J. Dong, Y. Zhao and T. Peng, "Architecture and Design Pattern Discovery Techniques – A Review," *Proc. International Conference on Software Engineering Research and Practice (SERP)*, 2007.

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

[9] H.C. Lovatt, A.M. Sloane and D.R. Verity, "A pattern enforcing compiler (PEC) for Java: using the compiler," *Proc. Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)* 2005, pp. 69-78.

[10] A. Blewitt, A. Bundy and I. Stark., "Automatic verification of design patterns in Java," *Proc. 20th IEEE/ACM international Conference on Automated Software Engineering (ASE '05)*, 2005.

[11] M. Eichberg, S. Kloppenburg, K. Klose and M. Mezini, "Defining and continuous checking of structural program dependencies," *Proc. 30th international conference on Software engineering (ICSE '08)*, 2008.

[12] C. Zhao, J. Kong and K. Zhang, "Design Pattern Evolution and Verification Using Graph Transformation," *Proc. 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007.

[13] C. Zhao, J. Kongb, J. Donga and K. Zhang, "Pattern-based design evolution using graph transformation," *Journal of Visual Languages and Computing*, vol. 18, no. 4, 2007, pp. 378-398.

[14] D. Kim and W. Shen, "An Approach to Evaluating Structural Pattern Conformance of UML Models," *Proc. ACM symposium on Applied computing (SAC'07)*, 2007.

[15] H. Zhu, I. Bayley, L. Shan and R. Amphlett, "Tool Support for Design Pattern Recognition at Model Level," *Proc. 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, 2009, pp. 228-233.

[16] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," *Proc. International Conference on Software Maintenance (ICSM'03)*, 2003, pp. 305 - 314.

[17] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, vol. 83, no. 2, 2010, pp. 209-221.

[18] N. Tsantalis, A. Chatzigeorgiou and G. Stephanides, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering* vol. 32, no. 11, 2006, pp. 896 - 909.

[19] M. Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 30, no. 12, 2004, pp. 904-917.

[20] A.D. Lucia, V. Deufemia, C. Gravino and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 32, no. 7, 2009.

[21] G. Antoniol, G. Casazza, M.D. Penta and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, 2001, pp. 181-196.

[22] J. Dong, S. Yang and K. Zhang, "Visualizing Design Patterns in Their Applications and Compositions," *IEEE Transactions on Software Engineering*, vol. 33, no. 7, 2007, pp. 433 - 453.

[23] D. Jain and H.J. Yang, "Object Oriented Design Patterns, Design Structure, and Program Changes: An Industrial Case Study," *Proc. IEEE International Conference on Software Maintenance (ICSM'01)*, 2001.

Maen Hammad is an Assistant Professor in Software Engineering Department at The Hashemite University, Jordan. He completed his Ph.D. in computer science at Kent State University, USA in 2010. 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 maintenance, program comprehension and mining software repositories.

Mustafa Hammad is an Assistant Professor at Information Technology department in Mu'tah University, Al Karak - Jordan. He received his PhD. in computer science from New Mexico State University, USA in 2010. He received his Masters

degree in computer science from Al-Balqa Applied University, Jordan in 2005 and his B.Sc. in computer science from The Hashemite University, Jordan in 2002. His research interest is Software Engineering with focus on static and dynamic analysis and software evolution.

Ahmed Fawzi Otoom is currently working as an assistant dean at the Faculty of Prince Al-Hussein bin Abdullah II for Information Technology. He is also an assistant professor at the Software Engineering department at Hashemite University, Jordan. He has a PhD degree in computer science from the University of Technology, Sydney (UTS), Australia, 2010. In 2003, he received his master's degree in software engineering from the University of Western Sydney, Australia. In 2002, he received his bachelor degree in computer science from Jordan University of Science and Technology, Jordan. He worked as a lecturer at Jerash Private University, Jordan between 2003 and 2005. His main research interests include computer vision and pattern recognition techniques for image and video analysis with a focus on realistic scenarios within video surveillance.

Mohammad Bsoul is an Associate Professor in the Computer Science department of The Hashemite University. He received his B.Sc. in Computer Science from Jordan University of Science and Technology, Jordan, his Master from University of Western Sydney, Australia, and his Ph.D. from Loughborough University, UK. His research interests include wireless sensor networks, grid computing, distributed systems, and performance evaluation.