Design Patterns Consideration in Class Interactions Prediction Development

Nazri Kama[1,2], Tim French[2], Mark Reynolds[2]
[1]*Advanced Informatics School, Universiti Teknologi Malaysia*
nazrikama@citycampus.utm.my
[2]*Computer Science and Software Engineering, The University of Western Australia*
{nazri, tim, mark}@csse.uwa.edu.au

Abstract

Customer changing their needs is a typical phenomenon in the software development phase. Predicting impact of a change is needed prior to actual change implementation so that an effective planning can be made. Several high level artifact analysis approaches have been developed including performing the prediction using class interactions. Since design patterns are one of the elements that affect the structure of actual class interactions, this paper proposes a new class interactions prediction approach that considers a design pattern analysis in its process. To demonstrate the design pattern analysis implementation, this paper selects the Boundary-Controller-Entity (BCE) pattern as an instance. A comparison between the new approach (with design pattern analysis) and two selected current approaches (without design pattern analysis) were conducted. The contributions of the paper are: (1) a new class interactions prediction approach; and (2) an evaluation that shows the new approach gives more accurate class interactions prediction than the selected current approaches.

Keywords: *impact analysis; requirement interactions; class interactions; pattern; traceability; requirement; class.*

1. Introduction

A typical problem when developing a software is that when changes happen to any part of the software, it may produce unintended, expensive or even disastrous effects [1,2]. Change impact analysis or impact analysis has been used to manage these problems [1-3]. According to Bohner [1], impact analysis is an activity of assessing the effect after making changes to a software system. However, some researchers extend this activity from assessing the effect after making changes to assessing effect before making changes to the software [3-8]. This activity is known as a predictive impact analysis.

The predictive impact analysis is partitioned into two categories which are low level analysis and high level analysis. The low level analysis category focuses on predicting a change impact based on low level artifacts analysis e.g., class interactions analysis [3-5]. The class interactions analysis provides reasonable accurate results since the class represent final implementation of user requirements. However, despite giving good results, class interactions analysis faces several challenges: (1) it requires detailed understanding of the system and its implementation; and (2) the amount of information to be analyzed can be so overwhelming that it may lead the analysis results to error [7,8]. As an alternative, the high level analysis category has been introduced [6-10]. This category performs change impact prediction using high level artifacts analysis e.g., high level model [6-10]. One of the high level models is a class interaction prediction [9,10]. Results by performing the change impact prediction on the

class interaction prediction are considered reflect to the low level analysis or actual class interactions analysis. Therefore, the accuracy of the class interaction prediction is critically important as it reflects the accuracy of the change impact prediction.

One of the techniques to develop the class interaction prediction is through reflection of significant object interactions in requirement artifacts [9-11]. The significant object is an object that has potential to be used as a class name in actual class implementation. Thus, the significant object is an object that has relationship with a class that has a similar name with the significant object name. This relationship is also called "traceability link" [13]. This technique is based on the precept that the interactions between significant objects in requirement artifacts reflect the actual class interactions in coding artifacts. However, this technique faces a challenge when the software employs pattern or design pattern [11,12] in its implementation. This will make the reflection of the significant object interactions to class interactions is inaccurate as the design pattern class has no reflection with the significant object in requirement artifacts. For example in Boundary-Controller-Entity (BCE) pattern [11], this pattern does not allow any interaction between Boundary and Entity classes. It creates a Controller class that acts as a mediator class to manage interactions between these classes. Since the Controller class is independently created to support the pattern implementation, this class has no reflection with any significant object in requirement artifacts. This situation leads to inaccurate reflection results.

To support the above challenge, we introduce a new class interactions prediction approach that includes pattern consideration in its prediction process. This approach composes two main steps which are reflecting the significant object interactions to predict actual class interactions and performing pattern analysis. As a preliminary work, we have selected the BCE pattern as the pattern to be considered in the prediction process.

This paper is laid out as follows: Section 2 briefly describes the related work. Then, Section 3 explains the design pattern concept and followed by a comprehensive description of the new class interactions prediction approach in Section 4. Subsequently, Section 5 and Section 6 present the evaluation strategy and evaluation results. Thereafter, Section 7 analyzes the results. Finally, Section 8 presents the conclusion and future works

2. Related work

There has not been much work related to development of class interactions or class diagram from requirement artifacts. There are two categories of class interactions development which are requirement artifacts analysis and non-requirement artifacts analysis. For the requirement artifacts analysis, typically noun and noun phrase keyword analysis has been used to reflect the class name in the design artifacts. Interactions between the keywords are then reflected to class interactions. Liang [14] proposes a use case goal analysis rather than a use case description analysis. Sharble and and Cohen introduce grammatical analysis [15]. There are two approaches for grammatical analysis which are data-driven which emphasis on information that the keyword possesses and responsibility-driven that focuses on services provided by the keyword. Premerlani [11] shows a structured approach using Unified Modeling Language (UML) object interactions diagram to build the object model.

For the non-requirement artifacts analysis, Bahrami [16] introduces "common class pattern" approach that is based on identification of various kinds of classes, of which a system will typically consist. Among the classes are physical classes, business classes, logical classes, application classes, computer classes and behavioral classes. Wirfs-Brock et al [17] show a Class-Responsibility-Collaborator (CRC) card that is used in brainstorming sessions.

Each developer plays one or more cards where new classes are identified from the message passing between the developers.

One of the major shortcomings in the current approaches is that the source of predicting actual class interactions uses the requirement artifacts only and there is no design artifacts involvement. According to [9,10,12], one of the important elements that affect the actual class interactions is the design patterns selection. Thus, this paper introduces a new class interactions prediction approach that improves the current approaches by considering design pattern analysis in its prediction process.

3. Pattern

A Pattern is an idea that has been useful in one particular context and will probably be useful in other contexts [12]. Patterns help software developer to identify combinations of architectural and/or design solution that have been proven to provide effective solutions in the past, and may provide the basis for effective solutions in the future.

There are two categories of patterns which are coarse-grained patterns and fine-grained patterns. The coarse-grained patterns help to solve a high level or architectural problem. Example of this pattern is the BCE or MVC patterns [12]. Conversely for the fine-grained pattern, it focuses on low-level software implementation problem and example of this pattern is Observer, Singleton and Facade pattern [12].

To demonstrate the impact of pattern to class interactions structure, we examine the BCE pattern [12]. The BCE pattern focuses on separating the application classes into three categories which are Boundary, Controller and Entity classes. The Entity class is a class that possesses data or business rules that access to and updates data from/to database. For the Boundary class, it renders the content of the Entity class and presents the content to user. Finally the Controller class is responsible for translating the interactions from Boundary class and passing it to the Entity class.

To show the impact of the BCE pattern to class interactions structure, we use a simple example of producing class interactions from a requirement description. The requirement description is "*the student enters user id and password at the login form. The system authenticates the user information and display main menu*". According to the requirement description, we develop two different class interactions structures. Figure 1 shows a class interactions structure that is extracted from the requirement description whereby Figure 2 shows a class interactions structure after considering the BCE pattern.

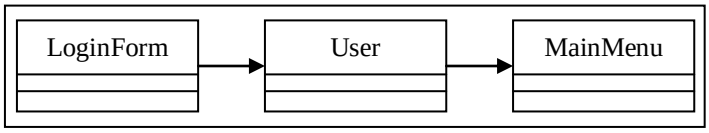

Figure 1. Before Applying the MVC Pattern

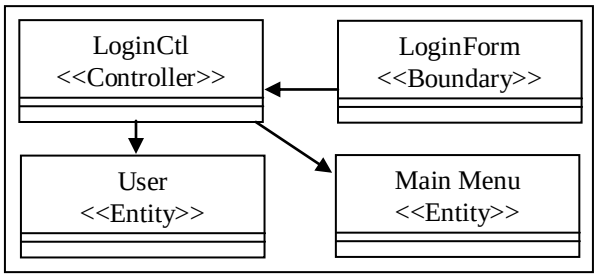

Figure 2. After Applying the MVC Pattern

Looking at the above figures, the different between the two class interactions structures is that in the BCE pattern there is no direct interaction between LoginForm class and User class. The BCE pattern uses Controller class to manage interactions between the LoginForm class and User class.

4. A new class interactions prediction approach

The following Figure 3 shows the overall structure of the new approach.

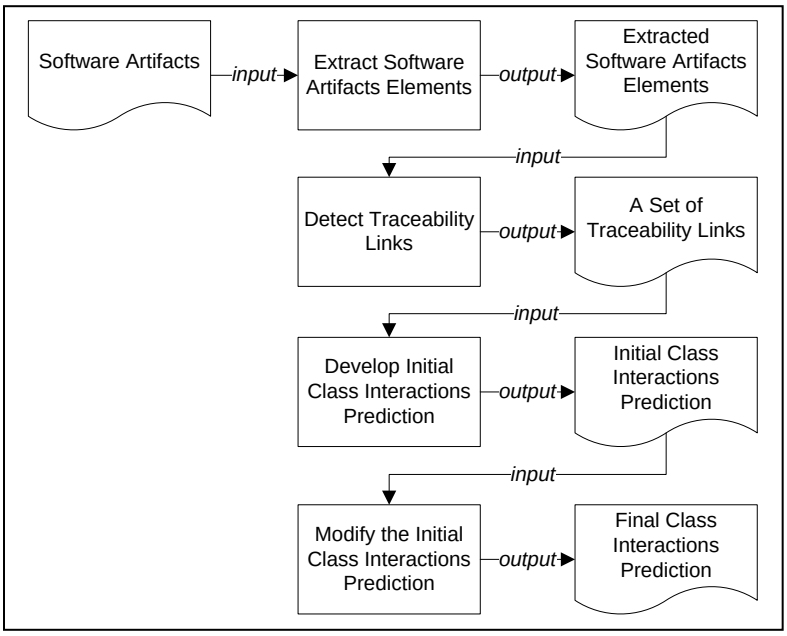

Figure 3. Class Interactions Prediction Approach

There are four processes in the new approach. The first process extracts software artifact elements from the software artifact i.e., requirement from requirement artifact, design class from design artifact and implementation class (which will be called "class" henceforth) from class artifact. The outcomes of the process are the extracted elements in each software artifact. The second process detects traceability links. There are two categories of traceability links which are the horizontal traceability link and the vertical traceability link. The horizontal traceability link detects traceability links across different software artifact (e.g., between requirement and design artifacts) whereby the vertical traceability links detects traceability links within a software artifact (e.g., among requirement artifacts). The outcome

of this process is traceability links among the extracted software elements. The third process develops an initial class interactions prediction based on reflection of significant object interactions in the requirement artifact. The outcome of this process is the initial class interactions prediction. Finally, the fourth process modifies the initial class interactions prediction based on design pattern. This process selects the BCE as the design pattern. The outcome of this process is the final class interactions prediction.

4.1. Process 1: Extract Software Elements in Software Artifacts

This process only extracts functional and non-functional requirements from the requirement artifact, design class properties (design class name and design class attribute name) from the design artifact and actual class properties (class name and class attribute name) from the class artifact. The extraction process is done manually by reviewing each of the software artifact documentation. Also, this process is commonly integrated in the requirement elicitation process.

4.2. Process 2: Detect Traceability Links

This process detects two categories of traceability link detections between the extracted software artifacts' elements. The categories are: (1) horizontal traceability links between requirements in the requirement artifact and the design class in the design artifact; and (2) vertical traceability links among requirements in the requirement artifact (requirement interactions). To detect the horizontal traceability links, a new technique that is based on the Rule-based technique is developed (we will refer to [18,19] as the "original technique"). The explanation and rationale of differences between the original technique and the new technique is summarized in Table 1 below.

Table 1. Modification Aspects on the Original Technique

No	Original Technique	The New Technique
1	There are two types of traceability detection rules which are the RTOM (requirement to object model) or horizontal traceability link type and the IREQ (inter requirement) or vertical traceability link type	The new technique modifies the RTOM type only. The IREQ type is excluded in the new technique
2	There are four types of RTOM traceability links which are: (1) Overlap link; (2) Requires_Execution_Of link; (3) Requires_Feature_In link; and (4) Can_Partially_Realise link	The new technique uses the Overlap and the Requires_Execution_Of type of traceability links only
3	There are two scenarios for the Overlap link	The new technique modifies the two existing scenarios and introduces four new scenarios for the Overlap link detection
4	There are three scenarios for the Requires_Execution_Of link	The new technique does not use the existing scenarios but introduces two new scenarios

The rational of modifications are described as below:

- Item [1]- The IREQ type is excluded in the new technique. The reason of exclusion is that in the original technique the IREQ rule implementation depends on the RTOM rule implementation. In other words if the RTOM rule implementation produces low accuracy of results, it will indirectly affect the accuracy of the IREQ rule implementation. Therefore, we have used our previous vertical traceabilty link detection technique that does not depend on the horizontal traceability link implementation. This technique is called Requirement Attributes Analysis [9].
- Item [2]- The Requires_Feature_In and the Can_Partially_Realise links are excluded in the new technique's implementation as they are used to support the IREQ rule implementation or to detect traceability link among requirement artifacts. Since in Item [1], the new technique uses the Requirement Attributes Technique, these two links are no longer practical to be included.
- Item [3]- The new technique modifies the two current scenarios of the Overlap link from performing a syntactic analysis between a sequence of terms in the requirement artifact and the elements in the analysis object model to the significant object name from use case specification and the elements in the design artifact. The reason of the modification is that the sequence of term tends to include other than noun in its analysis. The use of the significant object name extraction improves in the syntactic analysis implementation [10].
- Item [4]- The new technique does not use the original Requires_Execution_Of link detection as it uses/requires the UML operation stereotype signature i.e. <<set>> or <<get>>. Since the new technique does not use the UML class diagram to represent the design information, the original Requires_Execution_Of link detection is no longer reliable. The new scenarios will be described in Section 4.2.1.4.

4.2.1. Detect Horizontal Traceability Links between Requirements and Design Class Properties

A similarity analysis is used to detect traceability links between requirements and design class properties. This analysis compares between the extracted object names from the requirement artifact and the extracted design class properties (design class name and design class attribute name) from the design artifact. Prior to performing the comparison, the extracted object names are classified into three patterns of object name. These patterns of object name are used to assist the comparison analysis. If the comparison shows there is a similarity between these names, then a traceability link is established.

There are four steps in this technique which are: (1) Step 1: Extract potential significant object names from the requirement artifact; (2) Step 2: Classify the extracted potential significant object names into pattern of object names; (3) Step 3: Extract design class properties from the design artifact; and (4) Step 4: Detect traceability links between the extracted object names and the extracted design class properties.

4.2.1.1. Step 1: Extract Potential Significant Object Names from the Requirement Artifact. A potential significant object is an object that has responsibilities in achieving a requirement goal. The requirement goal is seen as either value that the requirement supplies

to its user of responses to its users and can be explicitly represented using a use case diagram [20]. The following Figure 4 shows the example of a requirement goal.

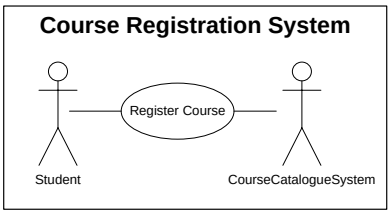

Figure 4. Example of a Requirement Goal Using a Use Case Diagram

The above figure describes the requirement goal aims to provide the student to register course in a Course Registration System. There are several potential significant objects that support the requirement's goal achievement such as a course or a student objects. Since these two objects are considered as the necessary or important information for a new course registration, then they are considered as the potential significant object names.

4.2.1.2. Step 2: Classify the Extracted Potential Significant Object Names into Pattern of Object Names: This step classifies the extracted potential significant object names from the requirement artifacts into pattern of object names. This classification is important as the similarity analysis will be conducted between the extracted potential significant object names with different pattern of object names and the design class name or the design class attribute name. There are three types of pattern of object names which are: (1) An object name that has a single noun (x). For example a "Student" significant object name; (2) An object name that has two nouns (x1, x2). For example a "Course Offerings" significant object name; and (3) An object name that has three nouns (x1, x2, x3). For example a "Course Catalogue System" significant object name.

4.2.1.3. Step 3: Extract Design Class Properties from the Design Artifact: This step extracts design class properties which are design class name and design class attribute name from the design artifact. These extracted names will be compared to the extracted significant object names in Step 4.

4.2.1.4. Step 4: Detect Traceability Links between the Extracted Significant Object Names and the Extracted Design Class Properties: This step detects traceability links by comparing the extracted potential significant object names and the extracted design class name and design class attribute name through two categories of similarity analysis. The first category is the similarity analysis between the potential significant object name and the design class name (SA#1) and the second category is the similarity analysis between the potential significant object name and the design class attribute name (SA#2). Detailed explanations of each similarity analysis are described below:

a. SA#1: Similarity analysis between the potential significant object name and the design class name

   This analysis compares the potential significant object name and the design class name. There are three similarity analysis guidelines which are: (1) SA#1.1: Similarity analysis between the potential significant object name with pattern of object name three nouns (*x1, x2, x3*) and the design class name; (2) SA #1.2: Similarity analysis between the potential significant object name with pattern of

object name two nouns (*x1*, *x2*) and the design class name and; (3) SA #1.3: Similarity analysis between the potential significant object name with pattern of object name a single noun (*x1*) and the design class name. Detailed explanations of each similarity analysis are described as below:

i. SA#1.1- Object Name (x1, x2, x3) vs. Design Class Name
This analysis compares the potential significant object name that consists of three nouns (x1, x2 and x3) and the design class name. If the analysis identifies any combination of a former noun qualifies and any latter noun in the potential significant object name with the design class name that contains both of them, then a traceability link is established. The combination can be between: (1) x1 and x3; (2) x1 and x2 or; (3) x2 and x3. Example of the SA#1.1 implementation in a typical Course Registration System: one of the extracted potential significant object names in the requirement artifact is a "Course Catalogue System" and one of the extracted design class name in the design artifact is "Course Catalogue". By implementing this rule, a traceability link is established between these names. This is because of a combination of Course (as x1) with Catalogue (as x2) is similar to the "Course Catalogue" design class name.

ii. SA#1.2- Object Name (x1, x2) vs. Design Class Name
This analysis compares the potential significant object name that consists of two nouns (x1 and x2) and the design class name. If the analysis identifies a combination of both nouns (x1 and x2) is similar with the design class name that contains both of them, then a traceability link is established. Example of the SA#1.2 implementation in the Course Registration System: one of the extracted potential significant object names in the requirement artifact is "Course Offerings" and one of the extracted design class name in the design artifact is "Course Offerings". By implementing this rule, a traceability link is established between these names. This is because of a combination of Course (as x1) with Offerings (as x2) is similar to the "Course Offerings" design class name.

iii. SA #1.3- Object Name (x1) vs. Design Class Name
This analysis compares between the significant object name that consists of a single noun (x1) and the design class name. If the analysis identifies the single noun (x1) is similar (or more than fifty percent similar) with the design class name, then a traceability link is established. Example of the SA#1.3 implementation in the Course Registration System: one of the extracted potential significant object names in the requirement artifact is "Professor" and one of the extracted design class name in the design artifact is "Professor". By implementing this rule, a traceability link is established between these names. This is because of Professor (as x1) is similar to the "Professor" design class name.

b. SA#2: Similarity analysis between the potential significant object name and the design class attribute name
This analysis compares between the potential significant object name and the design class attribute name. There are two similarity analysis guidelines which are: (1) SA#2.1: Similarity analysis between the potential significant object name with pattern of object name three nouns (x1, x2, x3) with the design class attribute name; and (3) SA#2.2: Similarity analysis between the potential significant object name with pattern of object name two nouns (x1, x2) with the design class attribute name. Detailed explanations of each type of analysis are described as below:

i. SA#2.1- Object Name (x1, x2, x3) vs. Design Class Attribute Name
This analysis compares the potential significant object name that consists of three nouns (x1, x2 and x3) and the design class attribute name. If the analysis identifies any combination of a former noun qualifies and any latter noun in the significant object name with the design class attribute name that contains both of them, then a traceability link is established. The combination can be between: (1) x1 and x3; (2) x1 and x2 or; (3) x2 and x3. Example of the SA#2.1 implementation in the Course Registration System: one of the extracted potential significant object names in the requirement artifact is "Registered Course ID" and one of the extracted design class attribute name in the design artifact is "Course ID". By implementing this rule, a traceability link is established between these names. This is because of a combination of Course (as x2) with ID (as x2) is similar to the "Registered Course ID" design class attribute name.

ii. SA#2.2- Object Name (x1, x2) vs. Design Class Attribute Name
This analysis compares the potential significant object name that consists of two nouns (x1 and x2) and the design class attribute name. If the analysis identifies a combination of both nouns (x1 and x2) is similar with the design class attribute name that contains both of them, then a traceability link is established. Example of the SA#2.2 implementation in the Course Registration System: of the extracted potential significant object names in the requirement artifact is "Course ID" and one of the extracted design class attribute name is "Course ID" from "Course" class. By implementing this rule, a traceability link is established between these names. This is because of a combination of the Course (as x1) with ID (as x2) is similar to the "Course ID" design class attribute name.

4.2.2. Detect Vertical Traceability Links in the Requirement Artifact or Requirement Interactions Detection: To detect the vertical traceability links, the requirement attribute analysis technique [9] is used. In brief, this technique introduces a set of requirement attributes that is tagged to each requirement description in the requirement artifact. The detection of requirement interactions is done by analyzing requirement attributes value using two requirement interactions detection guidelines. The reason of using this technique is that this technique not only capable to detect interactions among requirements like existing techniques [14,15] but also can be extended to detect the significant objects interactions in the requirement artifact (will be described in Section 3.3.1).

There are two steps in the technique which are: (1) Step 1: Restructure requirement into requirement attributes and; (2) Step 2: Detect requirement interactions using requirement interactions guidelines. In the first step, this technique restructures a requirement description into five requirement attributes. The attributes are: (1) Requirement Identifier (or ID)- A unique identification of each requirement in the requirement artifact; (2) Pre-condition (or Pre)- A list of condition(s) that a requirement that must fulfill prior to its execution; (3) Triggering Event (or TrE)- Action(s) that cause a requirement to be executed; (4) Requirement Description (or ReqtDescr)- Description of the original requirement from user. The description is broken into four sections; (4.1) Subject (Sub): Actor or user of the requirement; (4.2) Action (Act): Action that the Subject performs; (4.3) Target (Tar): Conceptual object or physical object such as hardware or external system that receives Action from the Subject; (4.4) Direction (Dir): Source or destination of the Target value; (5) Post-condition (or Post)- Result(s) after a requirement has successfully been implemented.

The second step detects requirement interactions using requirement interactions detection guidelines. There are two requirement interactions detection guidelines. The first guideline (G1) is (given R represents a requirement, SO represent a set of significant objects and SOS represent the significant object state such as displayed, calculated and etc.):

$$R2[[SO[Pre]] \text{ AND } [SOS[Pre]]] \cap R1[[SO[Post]] \text{ AND } [SOS[Post]]] = \emptyset$$

This guideline explains that if exists a significant object name and its state in the Pre-condition attribute of a requirement (R2) is similar to a significant object name and its state in the Post-condition attribute of another requirement (R1), then these two requirements are considered to have an interaction between them. The explanation is based on the precept that: "if R1 has a similar significant object name and its state in its Pre-condition attribute with the significant object name and its state in the Post-condition attribute of R2, it indicates that R2 and R1 interact between them".

The rationale of this guideline is that every requirement has pre-condition(s). This pre-condition must be validated whether it is successful or unsuccessful prior to the requirement execution. If the validation shows a successful result, then the requirement can be executed. Otherwise, the requirement will not be executed. There is another validation aspect in the pre-condition which is the validation of noun or significant object name state. Typically, the implementation of the noun or significant object name state is done by other requirements (or requirement description). After the other requirement has successfully implementing the noun or significant object name, the state results is stated in its post-condition. This scenario indicates that when a requirement (R2) that has a similar significant object name and its state in its pre-condition with the significant object name and its state in the post-condition of other requirement (R1), then R2 has potential to have interaction with R1.

The second guideline (G2) is:

$$R1[[[SO[Pre] \text{ AND } [SOS[Pre]]] \text{ AND } [SO[TrE] \text{ AND } [SOS[Pre]]]] \cap R2[[[SO[Pre] \text{ AND } [SOS[Pre]]] \text{ AND } [SO[TrE] \text{ AND } [SOS[Pre]]]] = \emptyset$$

This guideline explains that if exists a significant object name and its state in the Pre-condition and the significant object name and its state in the Trigger Event of a requirement (R1) is similar to a significant object name and its state in the Pre-condition and the significant object name and its state in the Trigger Event of another requirement (R2), then these two requirements are considered to have an interaction between them. The explanation is based on the precept that: "if R1 and R2 have a similar Pre-condition and Trigger Event attributes, it indicates that R2 and R1 interact between them".

The rationale for this guideline is that every requirement has its own pre-condition and trigger event. As described in the first guideline (G1), the Pre-condition must be validated whether it is successful or unsuccessful prior to the requirement execution. The Trigger Event indicates an event that consists of a significant object names and its state which causes a requirement to be executed. If two requirements have a similar Pre-condition and Trigger Event, these two requirements are considered as having a similar state prior to its execution. This similar state makes the significant object(s) in the Target attribute of both requirement descriptions have indirect interactions between them (share the same significant object(s) and its state).

4.3. Process 3: Develop initial Class Interactions Prediction

This process develops initial class interactions prediction based on reflection of the significant object interactions in the requirement artifact. Since this process assumes class interactions are developed based on the design class interactions, the design class interactions indirectly represent the initial class interaction prediction. There are two steps in this process which are: (1) Step 1: Detect significant object interactions; and (2) Step 2: Reflect the significant object interactions to design class interactions. Detailed explanations of each step are described as follow:

4.3.1. Step 1: Detect Significant Object Interactions: There are two types of the significant object interactions. The first type is the significant object interactions in a requirement description and the second type is the significant object interactions in two interacting requirement descriptions. To detect these types of interactions, requirement description structure as described in the requirement attribute analysis technique [9] is used. In the first type of interactions detection, the Target and the Direction attributes in the requirement description structure are used. These attributes values in all requirement descriptions are reviewed. The following detection guideline is used (given SO represents significant object and → symbol represents interaction):

$$SO[Tar] \rightarrow SO[Dir]$$

This guideline explains that significant object in the Target attribute will have interaction with the significant object in the Direction attribute. This guideline is based on the precept that the significant object in the Target attribute will be described where it will be sent to or where it comes from in the Direction attribute. Thus, the significant object that exists in the Target attribute will has interaction with the significant object in the Direction attribute.

The second type of interactions detection, the Pre-condition, Post-condition and Target attributes in the requirement description structure are used. These attributes values in all requirement descriptions are reviewed. The following detection guideline is used (given R represents a requirement, SO represent a significant object and SOS represent the significant object state such as displayed, calculated and etc.):

$$\text{IF } R2[[SO[Pre]] \text{ AND } [SOS[Pre]]] \cap R1[[SO[Post]] \text{ AND } [SOS[Post]]] = \emptyset \text{ THEN } R2[SO[Tar]] \rightarrow R1[SO[Post]]$$

There are two conditions in this guideline. The first condition is the detection can only be performed on two interacting requirement descriptions (R2 requires R1). The second condition is the significant object in the Target attribute of R2 will have interaction with the significant object in the Post-condition attribute of R1. This guideline is based on the precept that the significant object in the Target attribute of R2 can only perform its activity after the significant object in the Post-condition attribute of R1 is verified (either true of false). This situation indirectly shows the significant object in the Target attribute of R2 has interaction with the significant object in the Post-condition attribute of R1.

4.3.2. Step 2: Reflect the Significant Object Interactions to Design Class Interactions: The reflection is based on the precept that the significant object in the requirement artifact reflects to the design class. Thus, the interactions between the significant objects also reflect to the interactions between the design classes.

To describe the reflection process from the significant object interactions to design class interactions, the following example is used. Assuming that there are three interacting significant objects (SO) which are SO1, SO2 and SO3 where: (1) SO1 interacts with SO2; (2) SO1 interacts with SO3 and; (3) SO2 interacts with SO3. Also, the traceability links exist between the significant objects with the design class (DCL) are: (1) SO1 trace to DCL1; (2) SO2 trace to DCL2; and (3) SO3 trace to DCL3. Based on the interactions and the traceability links, Figure 5 illustrates the reflection process.

SO Interactions Matrix

	SO1	SO2	SO3
SO1			
SO2	√		
SO3	√	√	

Traceability Links

SO Name	DCL Name
SO1	DCL1
SO2	DCL2
SO3	DCL3

DCL Interactions Matrix

	DCL1	DCL2	DCL3
DCL1			
DCL2	√		
DCL3	√	√	

Figure 5. Reflection of Significant Object Interactions to Design Class Interactions

Since this process assumes class interactions are developed based on the design class interactions, the design class interactions indirectly represent the initial class interaction prediction. The following Figure 6 shows the reflection of design class interactions to the initial class interactions prediction.

DCL Interaction Matrix

	DCL1	DCL2	DCL3
DCL1			
DCL2	√		
DCL3	√	√	

Traceability Links

DCL Name	CL Name
DCL1	CL1
DCL2	CL2
DCL3	CL3

CL Interactions Matrix

	CL1	CL2	CL3
CL1			
CL2	√		
CL3	√	√	

Figure 6. Reflection of Design Class Interactions to Class Interactions

4.4. Process 4: Modify the Initial Class Interactions Prediction

This process modifies the initial class interactions prediction according to the BCE design pattern. As described in Section 2, one of the important elements that affect the actual class interactions is the design patterns selection [9,10,12]. Thus, this process intends to improve the initial class interactions prediction through design patterns consideration. There are three steps in this process which are: (1) Step 1: Identify affected classes by the design pattern implementation; (2) Create a new design pattern class type in the initial class interactions prediction; (3) Step 3: Modify the initial class interactions prediction based on the design pattern implementation.

4.4.1. Step 1: Identify Affected Classes by the Design Pattern Implementation: There are two types of classes that will be affected in the initial class interaction prediction. The types are the Boundary class and the Entity class. To identify these types of classes in the initial class interaction prediction, the role of each class is reviewed. For the Boundary class, there are three potential roles which are: (1) a class that communicate with human user of a software system that is also known as user interface class; (2) a class that communicate with other systems or external system that is also known as system interface class and; (3) a class

that communicates with devices to detect external events that is also known as hardware interface class. For the Entity class, there are two potential roles which are: (1) a class that stores information and; (2) a class that performs business logic or control of information processing.

4.4.2. Step 2: Create New Design Pattern Class in the Initial Class Interactions Prediction: A new class which is the Controller class is created per use case. The reason of the creation per use case is because of the Controller class is responsible for managing a use case implementation [17]. For example if the software system has five use cases, then five new Controller class are created in the initial class interactions prediction accordingly.

4.4.3. Step 3: Modify the Initial Class Interactions Prediction Based on the Design Pattern Implementation: Prior to modifying the initial class interactions prediction, all classes (Boundary, Controller and Entity classes) are reviewed to identify which use case they are belonged to. For the Boundary and the Entity classes, they are reviewed to identify which significant object in requirement descriptions they belonged to or are created from. The identification is done by reviewing the horizontal traceability links of these classes with the significant objects in requirement descriptions. However in some circumstances, these classes may be created from a significant object that exists in different use cases. If this situation occurs, it means that these classes belong to and can be classified in any of those use cases. For the Controller class, since this class is created per use case, then they automatically belongs to the use case that they are created from.

To modify the initial class interactions prediction, the following Table 2 shows the BCE design pattern interactions rules in a use case.

Table 2. BCE Design Pattern Interaction Rules

	Boundary	**Controller**	**Entity**
Boundary	Valid	Valid	Invalid
Controller	Valid	Valid	Valid
Entity	Invalid	Valid	Valid

According to the above table, there are two significant interaction rules in the BCE design pattern: (1) The BCE design pattern does not allow interaction between the Boundary class and the Entity class. Any interactions between these classes are considered as invalid interaction and it needs to be eliminated; and (2) The BCE design pattern uses the Controller class as a mediator of interactions between the Boundary class and the Entity class.

Table 3 below shows the example of applying the process steps for the other two selected design patterns which are the Model-View-Controller (MVC) and the Broker design patterns [12]. In brief, the MVC design pattern intends to provide an ability of an application to maintain multiple views or presentations of a single data. This design pattern hinges a clear separation of class application into three types of design pattern classes. The classes are the Model class that is used for maintaining the application data, the View class for presenting all or portion of the Model class data and the Controller class for handling events that interact with the Model class and the View class.

For the Broker design pattern, this pattern focuses on structuring a distributed application to hide or to encapsulate implementation details of remote service invocation. The encapsulation is done by structuring the details of remote service invocation into a different layer than the business logic layer. This layer provides an interface that allows client class to

invoke any methods like any local interface from the server class. This pattern considers any classes that use the remote service invocation service as a client class whereby any classes that respond to the service as a server class. However, detailed implementation of the MVC and the Broker pattern are not considered in detail here.

Table 3. Summary of the MVC and Broker Patterns' Implementation

Step Description	Design Pattern	
	MVC Pattern	Broker Pattern
Step 1: Identify Affected Classes by the Selected Design Pattern Implementation	User interface class and business logic class. The user interface class is considered as the View class whereby the business logic class as the Model class	Any class or client class that requires service from any class that provide the service or server class
Step 2: Create a New Design Pattern Class Type in the Initial Class Interactions Prediction	Controller class	Server Class, Client Class and Broker Classes
Step 3: Modify the Initial Class Interactions Prediction According to the Selected Design Pattern Implementation	Eliminate existing interactions between the user interface class and the business logic class. Develop new Controller class interactions with the user interface class and the Model class	Eliminate existing interactions between the client class and server class. Develop new Server Proxy, Client Proxy and Broker Classes with the client and server classes

5. Evaluation Strategy

The evaluation aims to compare the accuracy of class interactions prediction produced by the new approach and selected current class interactions prediction approaches. There are four elements have been considered in the evaluation strategy which are case studies, evaluation process, evaluation metrics and threat to validity. The following sub-sections describe detailed implementation of each element.

5.1. Case Study

Five software projects were selected to evaluate the capability of the improved approach. These software projects were developed by several groups of final year post-graduate students of software engineering course at the Centre for Advanced Software Engineering, Universiti Teknologi Malaysia. The following Table 4 shows the software projects profiles.

Table 4. Software Project Profiles

ID	No of Reqts	No of Use Cases	No of Classes
P1	58	7	36
P2	50	6	38
P3	48	7	40
P4	53	6	39
P5	57	8	39

5.2. Evaluation Process

There are three steps in the evaluation process. The first step is to extract software artifacts documentations versions from each software project. There are two sets of the extracted documentations which are design phase and coding phase versions. The reason of extracting these versions is to validate the effectiveness of the all approaches (current approaches and the new approach) to develop class interactions prediction with and without design pattern class involvement. We assume that the design phase version consists of minimal design pattern class involvement as most design pattern class has not been developed yet. For the coding phase version, maximal design pattern class involvement where most of the design pattern classes have been developed. Each version consists of three types of software artifacts which are requirement artifacts and coding artifacts.

The second step is to develop class interactions prediction for each software project using selected current approaches and the proposed approach. We have selected two current class interaction prediction approaches which are Grammatical Analysis (GA) [14] and Use-Case Goal (UCG) [15]. Both approaches use reflection of object interactions concept to develop class interactions prediction. The main difference between the current approaches and the proposed approach is that the new proposed approach includes Pattern analysis in its prediction process.

The third step is to compare the current class interactions prediction approaches results with the proposed class interactions prediction approach results. We employed our previous developed set of evaluation metrics [9,10] to evaluate the accuracy of class interactions prediction.

5.3. Evaluation Metric

This study has employed an evaluation metric. Briefly, each generated class interactions prediction can be assessed according to four numbers: NP-NI (the number of pairs of classes correctly predicted to not interacting), P-NI (the number of pairs incorrectly predicted to interacting), NP-I (the number of classes incorrectly predicted to not interacting) and P-I (the number of classes correctly predicted to interacting). These numbers is then used to calculate a Kappa value [21], which reflects the accuracy or the prediction (0 is no better than random chance, 0.4-0.6 is moderate agreement, 0.6-0.8 is substantial agreement, and 0.8-1 is almost perfect agreement).

5.4. Threat to Validity

There were four threats identified during evaluating the proposed approach. The threats are: (1) Defect on the selected current class interaction prediction approaches implementation. We have gathered the current approaches information from published research works, especially from book chapters and conference proceedings. Some information in the sources

may contain brief and compressed information (due to space restriction). However we have made justification based on the precept that those sources were already published and well accepted in academic community, in term of their information 'completeness' and clarity; (2) Inconsistency between object names in requirement artifacts and actual class implementation. We have requested the software developer to review and update the software documentations accordingly prior to performing the class interactions prediction process; (3) Poor class interactions implementations. To avoid poor actual class interactions such as some interactions are against the BCE interactions rules, each software team is required to review and correct them accordingly. However, some of them are still there but with fairly minimal number of interactions across the software projects; and (4) various pattern of class implementation. This evaluation focuses on BCE pattern analysis. However, some projects create their own pattern such as database transaction pattern. This pattern manages interactions from Entity class to table in application database. All interactions between classes that involve in the database pattern are excluded in the prediction results.

6. Evaluation Results

The evaluation results are divided into two groups which are: (1) class interactions prediction results produced by current approaches (GA and UCG approaches) and; (2) class interactions prediction results produced by the proposed approach.

6.1. Results Produced by the Current Approaches (without Pattern Analysis): The following Table 5 and Table 6 show the prediction results produced by the GA and UCG approaches accordingly.

Table 5. Prediction Results Produced by the GA Approach

Attribute	Design Phase Version					Coding Phase Version				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
NP-NI	98	95	98	106	119	253	258	302	287	274
P-NI	7	6	4	6	5	102	135	112	105	143
NP-I	5	7	6	4	6	110	105	131	132	110
P-I	100	102	102	115	123	201	243	275	256	253
Corr	93	94	96	95	96	66	64	71	71	64
Com	95	94	94	97	95	65	70	68	66	70
Kappa	0.901	0.894	0.918	0.927	0.925	0.399	0.422	0.465	0.448	0.418

Table 6. Prediction Results Produced by the UCG Approach

Attribute	Design Phase Version					Coding Phase Version				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
NP-NI	94	90	101	102	110	244	284	297	287	276
P-NI	6	7	7	10	8	110	132	123	105	127
NP-I	8	6	9	6	6	104	98	114	132	123
P-I	102	107	93	113	129	208	227	286	256	254
Corr	94	94	93	92	94	65	63	70	71	67
Com	93	95	91	95	96	67	70	72	66	67
Kappa	0.887	0.898	0.862	0.883	0.909	0.408	0.415	0.487	0.448	0.423

6.1.1. Iteration 1- Design Phase Results (GA and UCG): Both approaches show: (a) the correctness and completeness values across software projects indicate a high prediction value where more than two-third of the actual class interactions were predicted and; (b) The kappa values show an almost perfect strength of agreement between the class interactions prediction and the actual class interactions.

6.1.2. Iteration 2- Coding Phase Results (GA and UCG): Both approaches show: (a) the correctness and completeness values across software projects indicate a low prediction value where less than two-third of the actual class interactions were predicted and; (b) the kappa values show the approaches produce a moderate strength of agreement between the class interactions prediction and the actual class interactions.

6.2 Results Produced by the Proposed Approach (with Pattern Analysis)

The following Table 7 shows the prediction results produced by the proposed approach.

Table 7. Prediction Results Produced by the Proposed Approach

Attribute	Design Phase Version					Coding Phase Version				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
NP-NI	98	95	98	106	119	276	303	338	335	344
P-NI	7	6	4	6	5	53	61	45	56	34
NP-I	5	7	6	4	6	68	62	52	33	46
P-I	100	102	102	115	123	269	315	385	356	356
Corr	93	94	96	95	96	84	84	90	86	91
Com	95	94	94	97	95	80	84	88	92	89
Kappa	0.901	0.894	0.918	0.927	0.925	0.678	0.713	0.802	0.805	0.823

Iteration 1- Design Phase and Iteration 2- Coding Phase Results: Both iterations results show (a) the correctness and completeness values across software projects indicate a high prediction value where more than two-third of the actual class interactions were predicted and; (b) Iteration 1 results: the kappa values show the approach produce an almost perfect strength of agreement between the class interactions prediction and the actual class interactions. Iteration 2: the kappa values show a substantial and an almost perfect strength of agreement in between the class interactions prediction and the actual class interactions.

7. Analysis of Results

Prior to analyzing the results, we construct the following Table 8 in order to show the number of developed design pattern classes.

Table 8. Number of Developed Design Pattern Class and Kappa Value

Attribute	Iteration 1- Design Phase					Iteration 2- Coding Phase				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
No of Developed Design Pattern Class	0	0	0	0	0	13	11	10	9	8
Kappa Value	0.9	0.9	0.9	0.9	0.9	0.4	0.4	0.5	0.5	0.4

7.3.1 Current Approaches Iteration 1 Results vs. New Approach Iteration 1 Results Analysis: Looking at Table 4, Table 5 and Table 6 results, all Kappa values indicate a high prediction value. Also, the results show there is no significant different between results produced by the selected current approaches and the new approach. Relating the accuracy of the prediction results to the number of design pattern class (see Table 6), we would like to explain that there is no clear indication that the number of design pattern class affects the accuracy of the prediction results. This is because of there is no design pattern class involvement in the prediction. Therefore, we make an argument that all approaches manage to produce high accuracy of prediction results when there is no design pattern class exist in the actual class interactions.

7.3.2 Current Approaches Iteration 2 Results vs. New Approach Iteration 2 Results Analysis: Looking at Table 4, Table 5 and Table 6 results, both Kappa values for the current approaches indicate a low prediction value whereby the new approach indicates a high prediction value. Relating the accuracy of the prediction results to the number of design pattern class (see Table 6), we would like to explain that there is a clear indication that the number of design pattern class in the actual class interactions affects the accuracy of the prediction results. This can be seen by looking at the current approaches results. The results show that the accuracy of the prediction results (Kappa value) are lower when design pattern class exists in the actual class interactions. Therefore, we make two arguments from this iteration results. The first argument is that the current approaches are not able to produce high accuracy of prediction results when there is design pattern class exists in the actual class interactions. The second argument is that the new approach gives a higher accuracy of prediction results than the current approaches when design pattern classes exist in the actual class interactions. This argument indirectly supports the importance of pattern consideration in the class interactions prediction process.

8. Conclusion and Future Work

We have proposed a new class interactions prediction approach that composes two main steps. The first step develops an initial class interactions prediction through reflection of significant object interactions in requirement artifacts via traceability analysis. The second step improves the initial class interactions prediction through design pattern analysis. The difference with the proposed approach and current approaches is that the proposed approach includes the Pattern analysis in its prediction process.

Besides introducing the new approach, we have compared the capability of the new approach to predict class interactions with two most popular selected current approaches. The evaluation results reveal that that the new approach gives better accuracy of prediction results than the selected current approaches. This evaluation results also indirectly show the importance of Pattern consideration in class interactions prediction process.

However, current application of the new approach restricts to a software application that employs Boundary-Controller-Entity pattern. The extension to other patterns will be investigated by future work. Furthermore, a demonstration of performing predictive impact analysis using the new class interaction prediction will be included as well.

References

[1] S. Bohner, and R. Arnold, "Impact Analysis - Towards a Framework for Comparison", Proceeding of the IEEE International Conference on Software Maintenance, IEEE Press, Washington USA, September 1993, pp. 292-301.

[2] R. J. Turver, and M. Munro, "An Early Impact Analysis Technique for Software Maintenance", Journal of Software Maintenance: Research and Practice, John Wiley and Son Ltd, Malden USA, February 1994, pp. 35-52.

[3] B. Breech, A. Danalis, S. Shindo, and L. Pollock, "Online Impact Analysis via Dynamic Compilation Technology", Proceeding of the 20th IEEE International Conference on Software Maintanence, IEEE Press, Illinois USA, 11-14 Sept. 2004, pp. 453--457.

[4] J. Law, and G. Rothermel, "Incremental Dynamic Impact Analysis for Evolving Software Systems", Proceeding of the 14th International Symposium on Software Reliability Engineering, IEEE Press, Colorado USA, 17-21 November 2003, pp 430.

[5] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing", Proceeding of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, IEEE Press, Helsinki Findland, 1-5 September 2003, pp. 128 – 137.

[6] Y. Li, J. Li, J, Y. Yang, and L. Mingshu, "Requirement-centric Traceability for Change Impact Analysis: A Case Study", Making Globally Distributed Software Development a Success Story, Springer Berlin/Heidelberg, 6 May 2008, pp. 100-111.

[7] J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli, "Change Impact Analysis for Requirement Evolution using Use Case Maps", Proceeding of the 8th International Workshop on Principles of Software Evolution, IEEE Press, Lisbon Portugal, 9 Jan 2006, pp. 81-90.

[8] M. Shiri, J. Hassine, and J. Rilling, "A Requirement Level Modification Analysis Support Framework", Proceeding of the 3rd International IEEE Workshop on Software Evolvability, IEEE Press, Paris, 1 October 2007, pp. 67-74.

[9] N. Kama, T. French, and M. Reynolds, "Predicting Class Interaction from Requirement Interaction", Proceeding of the 13th IASTED International Conference on Software Engineering and Application, ACTA Press, Cambridge USA, 2-4 November 2009, pp. 30-37.

[10] N. Kama, T. French, and M. Reynolds, "Predicting Class Interaction from Requirement Interaction: Evaluating a New Filtration Approach", Proceeding of the IASTED International Conference on Software Engineering, ACTA Press, Innsbruck Austria, 16-18 February 2010, 109-116.

[11] J. Premerlani, J. Rumbaugh, M. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, 1991.

[12] M. Fowler, Analysis Patterns – Reusable Object Models, Addison Wesley, Canada, 19 October 1996.

[13] A. Bianchi, A. R. Fasolino, and G. Visaggio, "An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models", Proceeding of the International Workshop on Program Comprehension, ACM Press, Washington USA, 10-11 June 2000, pp. 149-158.

[14] Y. Liang, "From Use Cases to Classes: A Way of Building Object Model with UML" Journal of Information and Software Technology, Elsevier, New York USA, 1 February 2003, pp. 83-93.

[15] R. C. Sharble, and S. S. Cohen, "The Object-oriented Brewery: A Comparison of Two Object-oriented Development Methods", Software Engineering Notes, ACM Press, New York USA, April 1993, pp. 60-73

[16] A. Bahrami, Object Oriented Systems Development, Tata McGraw-Hill, India, August 2008.

[17] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs USA, 28 June 1990.

[18] G. Spanoudakis, A. Zisman, E.P. Minana, and P. Krause, "Rule-based Generation of Traceability Relation", Journal of System and Software, Elsevier Inc., New York USA, July 2004, pp. 105-127.

[19] G. Spanoudakis, "Plausible and Adaptive Requirement Traceability Structures", ACM Press, New York USA, 15-19 July 2002, pp. 135-142.

[20] T. Quatrani, Visual Modeling with Rational Rose 2002 and UML, Addison Wesley, Englewood Cliffs USA, 19 December 1997.

[21] J. Cohen, "A Coefficient of Agreement for Nominal Scales", Journal of Educational and Psychological Measurement, Sage Press, London UK, April 1960, pp. 37-46.

Authors

Nazri Kama received his Master Degree in Real-time Software Engineering and Bachelor Degree in Management Information System from Universiti Teknologi Malaysia in 2002 and 2000 respectively. His research interests are in software development, software maintenance, impact analysis, traceability and requirement interactions.

Tim French received his PhD from the University of Western Australia in 2007 where he is currently the Coordinator of the Software Engineering Program. He is interested in formal methods for software engineer and applications of logic in computer science.

Mark Reynolds obtained his first degree at The University of Western Australia (UWA) in Pure Mathematics and Statistics in 1984, his PhD at Imperial College London (IC) in Logic Programming in 1988 and a Diploma in Education from UWA in 1988. He lectured in the Department of Computer Science at UWA in 1988 and 1989. From 1990 until 1995 he worked as Research Assistant then Research Fellow on various UK and European Research Council funded projects on temporal logic, coordination programming and specification of safety-critical systems. In July 1995, he became a lecturer at Kings College London and in July 1998, he moved to Murdoch University. After rejoining the CSSE school at UWA as an Associate Professor in 2004, he became a Professor in 2009. His main research interests are in the use of temporal logic and related formal methods in software engineering and he is one of the authors of a two-volume major research monograph on the mathematical foundations and computational aspects of Temporal Logic.