Analyzing Call Dependencies to Improve Automatically   
Detecting Outdated Requirements

1. Introduction
2. Outdated Requirements Detecting and Call Dependencies
3. Proposed Approach

This section presents our approach for detecting outdated requirements with call dependencies analyzing. The approach consists of 4 main steps (see Figure 1). First we identify the changes of code elements by comparing two versions of the source code (Section III-A). Second, we do call dependencies analyzing to construct change region based on the changed elements (Section III-B). Third, to describe each change region, we extract keywords from the code elements in change region to compose the text description (Section III-C). Finally, we use IR-based techniques to calculate the similarities between the descriptions of change regions and the requirement specification, and the candidate list of outdated requirements is generated according to their similarities (Section III-D).

1. Comparing Changes of Code Elements

When software evolves, not all changes in source code impact the requirements. In fact, many of the code changes are bug fixes, refactoring, changes in implementation details, and so on. Eya et al. [xx] proposed that the addition and deletion of code elements (packages, classes, methods or fields) are the changes which are likely to impact the external behavior of the system and thus requirements. They compared all the elements in the code to detect the ones that have been added and removed, then filtered out the additions and deletions that are due to renames. In this step, we followed their process in [xx] to detect changes of code elements.

Beyond that, we suppose that an added package is composed of few added classes and an added class is composed of few added methods and fields. Therefore, we first extend the added packages to added classes, then we extend the added classes to added methods and fields. The removed elements can be processed in similar way. To represent the changes of code elements in source codes, Let M = {m1, …, mn} be a set where each mi is an added or removed method and F = {f1, …, fn} where each fi is an added or removed field.

(First, we compared all the elements in the code to detect the ones that have been added and removed. The comparison is based on the name only, and the order of comparison is from high level elements to low level elements. Therefore, a package is considered as added(removed) to the new(old) version if there is no other package having the same name in the old(new) version. )

Beyond above

Let M = {m1, …, mn} be a set where each mi is an added or removed method and F = {f1, …, fn} where each fi is an added or removed field.

(We represent a change of code element as a triple {ElementID, ChangedType, })

(We suppose that a package can be composed of classes, and a class can be composed of methods and fields.)

1. Constructing Change Region
2. Creating CDCGraph (Call Dependency with Closeness Graph)  
   In this part, we use the CDCGraph to describe the call dependencies and their closeness of a source code version. Our algorithm for creating CDCGraph includes following two steps:
3. Establishing call dependencies. We define a CDGraph (Call Dependency Graph) as an ordered pair G = <V, E>, where vertexes V is a set of methods in source code and edges E is a set of call dependencies between the methods. By call dependencies we mean if a method A invokes another method B, then A depends on B. We use Apache BCEL to capture call dependencies from compiled jar file. For those projects which can’t be compiled directly, we invent a tool base on JDT to capture call dependencies from java files.
4. Calculating the closeness of call dependencies. In CDGraph, the call dependencies between two methods neither exists nor does not exist, and all existing call dependencies are considered equally tight. In fact, when a method A invokes another method B, the dependency should be strong if two methods are cooperating on the same task, or should be weak if it just indicates the switching between tasks. To measure the call dependencies, we use closeness to quantify the degree of interaction among methods. We define the following formula to calculate the closeness for call dependencies: (Closenesse = 2 / (OutDegreee.caller + InDegreee.callee) (1))  
   where OutDegreee.caller represents the out-degree of caller method in CDGraph and InDegreee.callee represents the in-degree of callee method.
5. because of two reasons: (1)
6. Establishing Initial Change Regions
7. Merging Initial Change Regions
8. Extracting Keywords for Change Region
9. Generating the Candidate List of Outdated Requirements by IR Techniques
10. Evaluation
11. Threats to Validity
12. Related Work
13. Future Work
14. Conclusions