periodica polytechnica

Electrical Engineering
54/3-4 (2010) 111–121
doi: 10.3311/pp.ee.2010-3-4.05
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

A comparison of programmers' opinions in change impact analysis

Gabriella Tóth

Received 2011-09-13

Abstract

Change impact analysis is generally regarded as a very difficult program comprehension problem. One of the reasons for this is that there is no universal definition for dependency between software artifacts, only algorithms that approximate the dependencies.

In the past two decades, different kinds of algorithms have been developed by researchers. But which algorithm is the most suitable in a specific situation, which one finds the relevant dependencies in the best way? What kinds of dependencies are important for the programmers? What kinds of algorithms do they work with? Finding the most relevant dependencies is difficult, and it is essentially a creative mental task.

A possible way to answer the above questions is to involve programmers in a survey, and listen to their subjective opinions based on expertise and experience in program comprehension. In this paper, we present the results of our survey on this theme. We wanted to know what the difference was between the results of some well-known algorithms and programmers' opinions and, in addition, among programmers' opinions as well. Hence we conducted a case study.

Keywords

Change impact analysis · software dependencies · JRipples · BEFRIEND

Gabriella Tóth

Department of Software Engineering, SzTE, H-6720 Szeged, Árpád tér 2, Hungary

1 Introduction

During its lifecycle, a software program can have various releases in which new features are added, bugs are fixed or new requirements are met. In order to implement such changes in the software, not only do the developers add new modules to the program, but they also alter the existing code itself to make it fit the new demands or requirements. Consequently, as the program evolves, more developers may be involved and the potential impact of change may not be fully understood by a developer making later modifications on the software. This is why impact analysis techniques are often used to determine the potential effect of a proposed software change on a subset or the whole of a given program [38].

Over time, different impact analysis algorithms have been developed. One of the most general groups is called computation-based algorithms, where the computational relationships between program elements are tracked, in particular via program slicing, call graphs and other similar methods. Another approach is to analyze different software artefacts in order to find possible semantic links e.g. a historically consistent co-change of program elements [17].

As observed by Boehm [34], modifying software generally involves three phases, namely understanding the existing software, modifying the software, and revalidating the modified software. In order to successfully complete a modification task, programmers must locate and understand the part of the software system that will be modified.

When the code to be understood is completely new to the programmer, Pennington [26] found that the programmers first build a control flow abstraction of the program called the *program model*. If the program model representation exists, he showed that a *situation model* is developed. This representation uses the program model to create a data-flow/functional abstraction. Then the *integrated model* assumes that programmers can start building a conceptual model at any level that appears suitable. Programmers can switch between any of the three model components during the comprehension process.

In this survey, we examined the thinking and approaches applied by different programmers during several impact analysis

tasks to generalize the programmers' decisions, and learn what kinds of dependencies they were able to identify. We used two different impact analysis tools, which present the potential dependencies in a different way. The impact sets found by using two different tools were compared and we examined the differences between programmers' impact sets based on their qualifications and experience as well.

The rest of the paper is organized as follows. In Section 2, we review related work on the comparison of programmers' views on program understanding. In Section 3, we continue a motivation example and draw up the main research questions. Then we overview the methods used in the study in Section 4. In Section 5, we describe the design and methodology of our study. Section 6 describes the results of the study, and Section 7 lists all the possible problems that might affect its validity. In the last section we draw some pertinent conclusions about the result of our survey.

2 Related work

Many empirical studies of programmers in software engineering have been reported in the literature. Basili *et al.* [27] laid the foundational work on this topic. They devised a framework for analyzing most of the experimental work performed in software engineering in the 1980s. They recommended that their framework be used to facilitate the definition, planning, operation, and interpretation of future studies as well. Moreover, the authors identified several key problem areas of experimentation in software engineering and discussed their possible solutions.

Ko *et al.* [30] presented a new model for program understanding as a process of searching, relating, and collecting information. Their goal was to investigate the programmers' strategies for understanding and utilizing relevant information and discover ways in which tools and environments (e.g. Eclipse) might be related to these strategies. There were two types of dependency navigations that they outlined; those based on direct and indirect static dependencies. In contrast to their approach, we looked for more kinds of static dependencies in the programmers' impact sets (direct and indirect dependencies as well) such as call, co-changing, slice and SEA relations.

Robillard *et al.* [31] not only looked at what developers do in general during a program modification task, but they also investigated what *successful* programmers do in contrast to *unsuccessful* ones. They concluded that in the context of a program investigation task, the systematic examination of a piece of code is generally more effective than the opportunistic approach. It tells us that algorithmic thinking is important in the understanding of a program, especially in the case of a modification task. We will try to identify the range of dependency methods available in programmers' impact sets.

Mosemann and Wiedenbeck [29] presented three methods for collecting information about the program: sequential, control flow, and data flow. Novice programmers were asked to use only one of these methods to understand a program. They found that the sequential and control flow methods of navigation were significantly different from the mean of the data flow navigation. Novice programmers had great difficulty in reading the program using just data flow navigation. However, there was no significant difference between the sequential and control flow methods. In our survey, we compared the impact sets computed by programmers and algorithms. While the programmers could access the code before and after the examined code part – so the sequential flow was given – , in our experiment, slice information consisted of control and data dependencies, while SEA and call information only consisted of control dependencies. So in this survey we examined this kind of methods and co-changing as well.

The above-mentioned papers are about program comprehension by programmers. However, our aim was also to retrieve different kinds of impact sets and compare them with the impact set identified by programmers. We found some papers that discussed a comparison of impact sets identified different.

Lindvall and Sandahl [12] sought to quantify how well experienced software developers predicted changes by conducting RDIA (Requirement-Driven Impact Analysis), where RDIA in their case was the general activity of predicting changes based on the change request. They compared and evaluated the predictions by examining the changes in the concrete implementation. Their results indicated that the impacted set predicted by the programmers without any help is generally correct but underestimated.

In one of our previous articles [32] we used the same data as in this paper. Then we presented an empirical comparison of four static impact analysis techniques (call information, static slice, SEA relation and co-changing files retrieved from SVN repositories) and dependencies which were determined by programmers. The paper reported an empirical study that focused on how much the different kinds of dependency sets support program comprehension. In that article we examined static impact analysis algorithms, not the programmers' opinions. Now we turn our attention to the programmers.

3 Motivation

There are several kinds of dependencies that may be found during a change impact analysis. Many algorithms exist which can determine the possible dependencies. In a certain situation it is important to know what kinds of dependencies are present. While some algorithms may provide irrelevant dependencies, some of them can be essential to propagate the given change. To illustrate the differences between impact sets computed by different algorithms, consider the example in Fig. 1.

When we compute the dependencies of these classes with different kinds of impact analysis algorithms, we may obtain quite different results. If we determine the impact sets of the particular classes with a call graph [21], we find that class `C` depends on class `A` and class `B` due to the method invocations. Using the static forward slice approach [33], we find that class `C` depends

```
class A{
  int x;
  public A(int i){
    x=i;
  }
  public int getX(){
    return x;
  }
}
```

```
class B{
  int y;
  public B(int j){
      y=j;
    }
    void setY(int i){
      y=i;
    }
}
```

```
class C{

  A a;
  B b;

  public void set(){
    b.setY(a.getX());
  }
}
```

Fig. 1. Motivating example.

on class A and class B due to control dependency and class B depends on class A because of data dependency. After by determining Static Execute After (SEA) relations [35], each class is related to each class.

This example shows how the behaviour of the impact analysis algorithms can vary. But what about the programmers? If the programmers try to collect all the relevant dependencies for a potential change requirement, what kinds of dependencies are they interested in?

Our **goal** is to contrast the impact sets identified automatically by algorithms with those identified manually by programmers. With the help of this study, we can get a deeper insight into a programmer's way of thinking and see what kinds of dependencies they find relevant or irrelevant in a given situation. We used two tools (JRipples [37] and BEFRIEND [45]) that can help the programmer to find the impact set of a given modification in two different ways: JRipples gives the potential dependencies in a step-by-step fashion through direct dependencies, while BEFRIEND shows the potential direct and indirect dependencies without the dependency path. Here we formulate the following research questions:

- **Q1:** In the case of determining dependencies in an incremental way by using JRipples, what proportion of dependencies identified by programmers are identified by the different impact analysis algorithms?
- **Q2:** In the case of determining dependencies from a set of direct and indirect dependencies by using BEFRIEND, what proportion of dependencies accepted by programmers were identified by the different impact analysis algorithms?
- **Q3:** Is there any difference among programmers' impact sets based on their qualifications and experience?

4 Impact analysis methods

During the evolution of a software program, programmers add new functionalities and release its new versions. If a programmer changes the source code, it may be difficult to determine the impact of the changes, especially in large applications, hence different tools (algorithms) are needed to handle this problem. In the survey we applied different impact analysis algorithms. In this section, we will overview the algorithms that were used.

The algorithms were implemented within the JRipples Java tool and framework [37, 41], which is an integrated tool in the Eclipse development environment supporting incremental change and relevant program analysis for the programmer, and it manages the organization of the steps that comprise the impact analysis and the subsequent change propagation. JRipples is based on the philosophy of 'intelligent assistance', which requires cooperation between the programmer and the tool itself. First, the tool analyzes the program, keeps track of any inconsistencies, and then automatically marks the classes/methods which should be visited by the programmer. Its main advantage is that it covers the algorithmic parts of the change propagation, which are often difficult or error-prone for humans.

Since it is straightforward to incorporate other analyzers (algorithms) into JRipples, it can serve as a framework for identifying several kinds of static dependencies. JRipples itself supports analysis on the granularity of classes and methods. In our experimental study, we determine dependencies on the granularity of methods (except historical co-change), but we raise the granularity to the class level for our comparison.

We implemented the following algorithms within JRipples:

- **Callgraph**. We built a directed graph that represents calling relationships between methods [21]. The graph was built based on AST computed by Eclipse JDT.
- **Static slice**. We apply static forward program slicing [33] (considering data and control dependencies) to determine the impact of the method modifications. The static forward slices were computed by the Indus Java static slicer API [23]. A slice is performed for each statement.
- **Static Execute After (SEA)**. According to the definition of SEA dependencies, method B depends on method A if and only if B may be executed after A in any possible execution of the program [35, 36]. The computation of SEA relations is an efficient analysis algorithm, which is able to produce conservative impact sets at the method level. The determination of these relations is based on the ICCFG representation [35] of

the program. We built the ICCFG graph based on AST computed by Eclipse JDT. We implemented the SEA algorithm on this graph and determined all the method pairs which are in a SEA relationship.

- **Co-changing files retrieved from SVN repositories**. Some dependencies are explicitly observed in the code; the software engineer only 'knows' which certain set of modules needs to be changed [40] to make a certain type of change. In such cases, to derive the set of source files impacted by a proposed change request, we can use historical data stored in a versioning system, namely Subversion (SVN). Since this way just the changed files can be retrieved, this analysis has only class granularity. A correlation value can be set between 0 and 1.0, if we would like to filter the co-changed classes found by the algorithms. For example, if the correlation value is 0.4, it means that class `A` depends on class `B` if in 40% of the commits when the `A` file changed, the `B` file changed as well. We got two result sets, one with a 0.4 correlation value and one with a 1.0 correlation value. We chose the correlation value of 40% because we did not want the union of the dependency sets to be overly large. The number of the SEA relations and the dependencies determined by programmers were the most extended and finding co-changing classes with a 40% correlation value gave the same amount of dependencies.

Altogether, we have 5 kinds of dependency sets (call, static forward slice, SEA, co-change$_{0.4}$, and co-change$_{1.0}$ relations). Two of these result sets have a method; two of them have a class; and one of them originally has a statement granularity but, of course, we raise all results to the class level to be able to compare them. Since the callgraph determines the call dependencies only one step at a time, we compute the transitive closure of the call relations of each method.

5 Description of the experiment

The experiment involving the participant programmers was performed in two stages (see Fig. 2). First, the participants were asked to use JRipples in 7 different use cases to discover the impact set of the hypothetical changes in some particular methods of our chosen sample project. Secondly, for each scenario we stored their results together with the results produced by the specific algorithms mentioned in the previous section in a common repository (BEFRIEND) that served as a control benchmark. Then we asked the participants to evaluate all of the stored dependencies individually. By doing this, we were able to calculate valuable statistics about the relationship between the different impact sets. The following subsections provide a detailed description of the above-mentioned stages and the preparation.

5.1 Subject project

First, we set up a test environment. We needed a sample project where the impact sets were defined according to the hypothetical modifications. When choosing the test project, we took the following into account:

- The code is written in Java, since JRipples analyzes only Java code.
- It has an accessible SVN repository with an extended history.
- It is a relatively complex, but not overly large piece of code – since the Indus slicer could not produce slices for large programs due to excessive memory consumption.
- It is compatible with JRE 1.4, since the Indus static slicer works only on this version of Java code.
- The code should be unknown to the participants, but be readily comprehensible.

Based on these requirements, we found an open source Java project called *ownSync*.[1] This is a small Java application that can synchronize two folders (on different machines or on the same machine) in both directions. The main characteristics of the sample project can be seen in Table 1.

Tab. 1. The characteristics of the subject system (ownSync).

No. of classes	No. of methods	LOC	Non-empty LOC	No. of commits
30	234	3666	3108	92

In this project, we defined some hypothetical change scenarios. We gave 7 methods from the sample project to the programmers so that they could examine them and determine the impact sets of their hypothetic changes. The methods were the following:

- *writeFolderState* of FolderState class,
- *internalMoveFile* of SyncTrashbox class,
- *loadConfig* of OwnSyncConfiguration class,
- *DeleteFolderAction* of DeleteFolderAction class,
- *forceDelete* of FileUtils class,
- *getSyncFolder* of FolderConfiguration class,
- *isAnyActionFailed* of OwnSyncResult class.

This selection was based purely on investigating the method types and their call information. Among them there are methods such as a constructor, a recursive, a getter method, a simple and a complex one, one which is called several times and one which calls several methods.

[1] http://sourceforge.net/projects/ownsync/

5.2 Participants

After selecting the sample project, 11 programmers with different qualifications and experience were invited to participate in our experiment. The group of participants consisted of 4 computer science students, 5 PhD students, and 2 software developers. Most of them have been working as software developer for years: they have experience in Java and program analysis as well. It is also interesting to note that the PhD students all attended an *Impact Analysis* course. From here on, the participants will be referred to simply as *programmers*.

5.3 Overview of the experiment

First, the list of the 7 methods from our sample project was given to each programmer and the task of the programmers was to apply JRipples to discover all of the methods impacted by any possible change made in these methods. As the starting point of the change (*concept location*) was found by the programmers with the help of JRipples, the remaining methods of the impact set were discovered in a step-by-step-look-at-the-neighbours fashion in the dependency graph built by JRipples.

We logged the programmers' actions to retrieve the dependencies, which were determined by the participants using JRipples. After everyone had finished their first stage task, the logs were collected. The log files contained the method level dependency for each scenario. Despite the fact that the programmers searched for dependencies at the method level, we raised the results to the class level.

The algorithms mentioned in Section 4 were implemented within JRipples and we collected all the class level impact sets for the same 7 criteria produced by the different algorithms. We got the union of all class level dependencies for each kind of impact set for all 7 methods.

Before the second stage, the union of the results, either found by a programmer or an algorithm, were provided together in BEFRIEND, whose database used in our experiment is publicly available online [45]. BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe) [42] is a general purpose benchmark tool. The benchmark was successfully applied for evaluating and comparing design pattern miner tools [44], clone detector tools [42], rule violation checkers, and now impact analysis algorithms. Although BEFRIEND is designed to be very general, some major improvements were required in order to make it capable of evaluating and comparing impact analysis results. After the BEFRIEND improvements, the union of the impact sets computed by algorithms or programmers was uploaded to the benchmark grouped by the scenarios.

In the second stage, the programmers evaluated the class level dependencies grouped by the scenarios without knowing which tool or programmer had found them. The programmers were asked the following question: 'Do you think there is a real dependency between these classes?' for each uploaded dependency. There were 4 possible answers to this question, which were

- Yes, I am sure that there is a dependency. (100%)
- I think there is a dependency. (66%)
- I think there is NO dependency. (33%)
- No, I am sure that there is no dependency. (0%)

We provided the opportunity of the evaluators not only to choose yes/no answers, but to describe their level of confidence. Each answer was associated with a percentage value forming a numerical scale from the firm negative answer through the solid negative and solid positive answer to the firm positive answer.

The second stage was complete when all the contributing programmers evaluated every single dependency. The outcome of the second stage yielded some valuable statistics that could be used as input for our key research questions.

6 Results and discussion

In this section, we supply concrete answers to our research questions. We calculated statistics from the different impact sets from Stage 1 and Stage 2. The two stages are very different: tools with different user interfaces and different confidence levels. In the first stage, the programmers determined the dependencies in a step-by-step fashion where only one piece of source code could be seen at the same time and only a *dependency* or *not a dependency* could be stated. Unlike JRipples, in the second stage BEFRIEND displays not only all the dependency candidates, but both sources of the classes of a certain dependency as well. BEFRIEND returns 4 possible values (0%, 33%, 66%, and 100%) in order to characterize the programmers' level of confidence.

6.1 Q1: In the case of determining dependencies in an incremental way by using JRipples, what proportion of dependencies identified by programmers are identified by the different impact analysis algorithms?

In the first stage, the programmers used the JRipples tool where they could identify dependencies incrementally via the dependency graph. They had to decide whether potential dependencies were really dependencies or not. The difficulty in using JRipples is that the potential dependencies appear step-by-step and the programmers can follow only one step graphically and they must keep the former dependencies (the path of origin) in their mind so as to think in a transitive way. The programmers can easily lose track of some information or overlook something important.

The algorithms and the programmers recognized 118 dependencies in total. Table 2 lists how many dependencies were determined by the given algorithms and the given programmers.

Table 3 shows the percentage of the dependencies identified by the given programmer obtained from the impact sets of different algorithms. There are dependencies that are not identified by any algorithms and their values can be seen in the last row (*others*). The sum of the rates in a column is over 100% due to

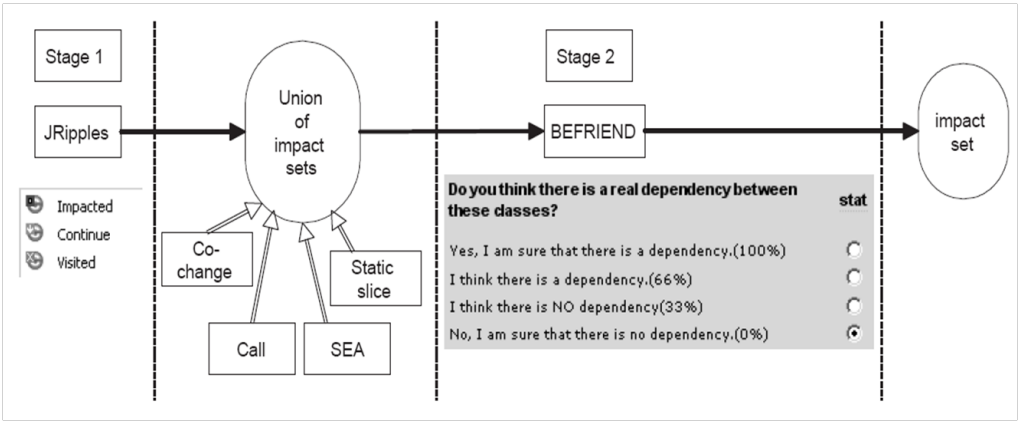

Fig. 2. An overview of the empirical study.

Tab. 2. The number of dependencies identified by the algorithms and the programmers.

Agent	No. of identified dependencies
call	15
slice	13
SEA	62
co-change$_{1.0}$	4
co-change$_{0.4}$	37
programmer$_1$	19
programmer$_2$	14
programmer$_3$	39
programmer$_4$	15
programmer$_5$	26
programmer$_6$	6
programmer$_7$	17
programmer$_8$	36
programmer$_9$	27
programmer$_{10}$	12
programmer$_{11}$	13

the overlapping parts of the impact sets (e.g. the SEA impact set contains the call impact set). Since the SEA and co-change$_{0.4}$ algorithms provide the most extended impact sets, this is why the values in these rows are the highest. SEA relationships cover the programmers' impact sets very well. But it should not be forgotten that slice and call relations are SEA relations as well. If we look at the average values, on average, most of the dependencies come from the SEA impact set, then co-change$_{0.4}$, callgraph, slice and finally co-change$_{1.0}$.

Considering that in Table 2 programmer$_3$ and programmer$_8$ found the most dependencies, among their dependencies there are a lot of *other* dependencies which were not determined by any algorithms used. While the SEA algorithm found the most dependencies, these two programmers found only about half of these dependencies. By contrast, programmer$_6$ found the fewest dependencies but all of them were SEA dependencies as well.

We can also view the dependencies identified by the programmers from another aspect; namely what proportion of the dependencies identified by the given algorithms were identified by the given programmers. Perhaps the call relations are the most easily recognizable. Table 4 shows that three programmers identified 40% of the call dependencies, which is the highest value on that row. However, programmer$_4$ did not identify any call relations, only SEA and co-change relations. It seems that the method to be modified was not examined by him, only those methods that are executed after it. Only 13% of SEA relations were identified by him because he was only looking for direct dependencies. Since programmer$_3$ found the most dependencies, he best covered the given kinds of relations. On average, almost the same percentage value of all the call, SEA and cochange$_{0.4}$ were identified by the programmers; they are about 20 - 22%. The slice dependencies were not so widely identified. Based on the higher call and SEA values, the control dependencies were considered, but the data dependencies were neglected.

6.2 Q2: In the case of determining dependencies from a set of direct and indirect dependencies by using BEFRIEND, what proportion of dependencies accepted by programmers were identified by the different impact analysis algorithms?

In the second stage the programmers had to determine dependencies again, but this time using the BEFRIEND tool. Here they had to decide whether a set of potential dependencies were really dependencies or not. The programmers could see all of the dependencies grouped by the scenarios, along with the direct and the indirect dependencies. The programmer could see the start and the end point of the dependency path, but the path was missing. Here we regard a potential dependency as a real dependency if the programmer's vote was at least 66%.

As seen in Figure 3, in most cases the programmers found more dependencies in the second stage than in the first stage. One reason for this difference is that in the second stage, we treat a dependency as a real dependency if the programmers' votes are at least 66%, while with JRipples they can only say whether it is dependency or not. So the accepted dependencies in the second stage consist of dependencies with less than 100% confidence level as well. Another reason is that in the first

Tab. 3. The distribution of the different kinds of dependencies among the dependencies identified in Stage 1.

	pr. 1	pr. 2	pr. 3	pr. 4	pr. 5	pr. 6	pr. 7	pr. 8	pr. 9	pr. 10	pr. 11	avg.
call	5%	21%	15%	0%	23%	33%	18%	11%	22%	8%	31%	17%
slice	0%	7%	8%	0%	15%	17%	6%	8%	7%	0%	15%	8%
SEA	84%	79%	56%	53%	58%	100%	94%	47%	52%	75%	100%	73%
co-change$_{1.0}$	5%	0%	3%	7%	4%	0%	0%	3%	4%	0%	0%	2%
co-change$_{0.4}$	32%	50%	28%	33%	35%	33%	35%	28%	37%	42%	38%	36%
others	5%	7%	31%	40%	31%	0%	0%	42%	33%	8%	0%	18%

Tab. 4. The percentage of all the call, SEA, slice, co-change dependencies identified by the programmers in Stage 1.

Type	pr. 1	pr. 2	pr. 3	pr. 4	pr. 5	pr. 6	pr. 7	pr. 8	pr. 9	pr. 10	pr. 11	avg.
call	7%	20%	40%	0%	40%	13%	20%	27%	40%	7%	27%	22%
slice	0%	8%	23%	0%	31%	8%	8%	23%	15%	0%	15%	12%
SEA	26%	18%	35%	13%	24%	10%	26%	27%	23%	15%	21%	22%
co-change$_{1.0}$	25%	0%	25%	25%	25%	0%	0%	25%	25%	0%	0%	14%
co-change$_{0.4}$	16%	19%	30%	14%	24%	5%	16%	27%	27%	14%	14%	19%

stage, JRipples represents dependencies in a step-by-step fashion, while in the second stage the programmers can see all of the dependencies for a scenario (dependencies that were identified by algorithms or programmers in the first stage). In the first case the programmer can follow only one step graphically and they must keep the former dependencies in mind in order to think in a transitive way. However, with BEFRIEND, they see the transitive dependencies and they only have to examine the given dependency and not keep information in their mind from an earlier step. The programmers can easily lose track of some information or overlook important information in the first stage.

Not surprisingly, we see that the programmers identified more dependencies in the second stage than in the first stage. We determined the intersection and the differences between the dependency sets identified in the first stage and the second stage for each programmer. In Table 5, the values in the table have been normalized by the size of the union of the given sets. According to this table, as a general rule, we can say that a few dependencies from the impact sets were missing, but several appeared in the second stage. We can see that programmer$_8$ found the same number of dependencies in both stages. But they are not the same dependencies; only 50% of these dependencies are the same, a quarter of the dependencies are absent and a quarter of the dependencies are new, so the contents of the set changed.

More precisely, the dependencies that were identified in the first stage, but rejected in the second stage are mainly SEA relations. This is due to the visual presentation of the tools: with JRipples the programmer could follow the dependency path, but with BEFRIEND to identify a two-or-more-unit-distance SEA relation is much more difficult. In the first stage, the programmers found a relatively large amount of SEA relations, some of which were rejected in the second stage. In contrast, the other kinds of dependencies (call, slice, co-change) were identified in smaller numbers in the first stage, so in the second stage the programmers were easily able to accept these kinds of new dependencies.

Table 6 shows that in the second stage the impact sets got by the programmers contain a higher percentage of dependencies identified by algorithms. The reason could be that if the programmers can see the possible dependencies together, they can examine them individually and it is easier to decide whether it is really a dependency to at least a 66% confidence level.

The SEA relations are present in smaller amounts, despite the increased number of identified dependencies. There are more identified dependencies, and the proportion of SEA relations is lower. The programmers did not discover new SEA relations. This may be because of the visual presentation of the BEFRIEND tool, and the indirect dependencies with a greater distance cannot be so easily identified based on the start and the end points.

On average, the amount of the dependencies identified only by programmers and not by any algorithm is the same, although the values got by the given programmers are not the same as in the first stage. The lower values increased and the higher values decreased. In the case of higher values, the dependencies not identified by any algorithms were simply absent. If the programmer did not find so many other dependencies in the first stage, they only accepted some dependencies identified by other programmers.

Although in the second stage the dependencies identified by an algorithm were in the potential dependencies which they had to examine, they identified many more dependencies that had been identified by one or more algorithms/programmers. Table 7 tells us that the scores are higher than in the first stage. The rank of the average values is different as well. Call relations were covered the best (49% of the call relations were identified on average); this kind of dependency can be recognized relatively easily, especially with the help of the recommendations by those programmers who identified more call relations. The data and control dependencies identified came to 36% of cases, while in

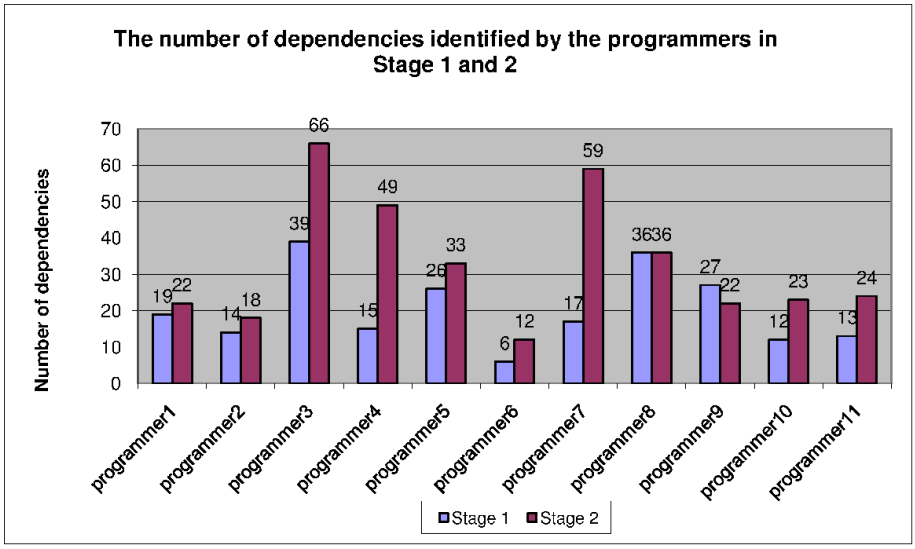

Fig. 3. The number of dependencies identified by the programmers in Stage 1 and Stage 2.

Tab. 5. Comparison of dependency sets identified by different algorithms.

	Stage 1∩Stage 2	Stage 1\Stage 2	Stage 2\Stage 1
$programmer_1$	64%	12%	24%
$programmer_2$	33%	25%	42%
$programmer_3$	46%	8%	46%
$programmer_4$	16%	11%	73%
$programmer_5$	69%	6%	26%
$programmer_6$	29%	14%	57%
$programmer_7$	29%	0%	71%
$programmer_8$	50%	25%	25%
$programmer_9$	36%	39%	25%
$programmer_{10}$	46%	4%	50%
$programmer_{11}$	28%	17%	55%

40% co-changed classes they were identified in 33% of cases.

6.3 Q3: Is there any difference among programmers' impact sets based on their qualifications and experience?

As we mentioned earlier, the participants are PhD students, computer science students and developers. There were 2 female PhD students ($programmer_7$ and $programmer_9$), while the others were males. $Programmer_3$, $programmer_5$, $programmer_7$, $programmer_8$, and $programmer_9$ were PhD students, $programmer_1$ and $programmer_4$ were developers, and $programmer_2$, $programmer_6$, $programmer_{10}$ and $programmer_{11}$ were computer science students.

According to Table 5, $programmer_3$, $programmer_5$, $programmer_7$, $programmer_8$, and $programmer_9$ covered the results of the algorithms better than the rest. These participants were the PhD students. Fig. 4 contains the average values in Table 5 for participants grouped according to qualifications and experience. This diagram also tells us that the PhD students covered best the different kinds of dependencies.

In Table 3 and Table 6 the developers and the students found more SEA relations, which means that in practice they consider SEA relations more useful than the PhD students. If a programmer has less background theoretical knowledge of impact analysis algorithms, the classes which are statically executed after the examined class are the most helpful during program comprehension or impact analysis. The PhD students are familiar with impact analysis algorithms, and about half of their identified dependencies are SEA relations, while *other* relations were identified with a higher score. The PhD students must have had other information – like comments and method with the same body – to go on in their final assessments.

7 Threats to Validity

This paper is an empirical study, and it has limitations that must be taken into account when evaluating the results and generalizing the findings to other contexts.

First, our hypothetical change requests may not equally represent real maintenance situations. If a programmer has a maintenance task, there is a certain program point in a method which needs some modification. By contrast, in our experiment the programmer had to find all the methods/classes impacted by *any* changes of the forward defined methods. This is necessary because the algorithms have a method or class granularity, not a statement granularity. However, for the testers this may repre-

Tab. 6. The distribution of the different kinds of dependencies among the dependencies identified in Stage 2.

Type	pr. 1	pr. 2	pr. 3	pr. 4	pr. 5	pr. 6	pr. 7	pr. 8	pr. 9	pr. 10	pr. 11	avg.
call	9%	44%	20%	20%	24%	25%	15%	14%	27%	35%	38%	25%
slice	0%	33%	11%	20%	12%	17%	8%	8%	18%	17%	29%	16%
SEA	73%	83%	33%	43%	67%	83%	63%	53%	68%	78%	71%	65%
co-change$_{1.0}$	5%	0%	3%	4%	3%	0%	5%	3%	9%	0%	0%	3%
co-change$_{0.4}$	36%	28%	35%	29%	36%	33%	39%	39%	50%	43%	42%	37%
others	14%	6%	33%	33%	21%	8%	22%	28%	18%	9%	8%	18%

Tab. 7. The percentage values of the all call, SEA, slice, co-change dependencies identified by the programmers in Stage 2.

Type	pr. 1	pr. 2	pr. 3	pr. 4	pr. 5	pr. 6	pr. 7	pr. 8	pr. 9	pr. 10	pr. 11	avg.
call	13%	53%	87%	67%	53%	20%	60%	33%	40%	53%	60%	49%
slice	0%	46%	54%	77%	31%	15%	38%	23%	31%	31%	54%	36%
SEA	26%	24%	35%	34%	35%	16%	60%	31%	24%	29%	27%	31%
co-change$_{1.0}$	25%	0%	50%	50%	25%	0%	75%	25%	50%	0%	0%	27%
co-change$_{0.4}$	22%	14%	62%	38%	32%	11%	62%	38%	30%	27%	27%	33%

sent a real maintenance situation: if the developers just supply the names of the changed methods to the testers, they must proceed from these methods to determine their impact sets and the necessary test cases.

There is another factor which is uncommon in software maintenance: the programmers are not familiar with the test project. Not every participant knows all the projects equally well, hence we chose a project which was not known to any of them. And here, the participants were all computer science students, PhD students or software engineers. Most of them were not familiar with impact analysis, but were common programmers who identified dependencies to the best of their knowledge and experience.

We considered algorithms that found impact sets for only one program. Other constraints were also mentioned in Section 5 (only Java code, memory consumption limitation, extended SVN history, etc.). In this case, the programmer can understand the project much better despite only having a partial understanding of several projects. Nevertheless, this project is an actual, non-trivial software system with a real SVN history. While only one subject system was examined, the empirical study has low statistical predictive power. We cannot claim that the results are generalizable.

When we compared the programmer evaluations, we noticed that they insisted on their previously observed dependencies, so it is possible that they remembered their opinions from the first stage. The solution to this might be to split the programmers into two groups and one of the groups determines dependencies only in the first stage, while the others do it only in the second stage.

8 Conclusions

Here we presented a detailed empirical comparison of impact sets determined by programmers and impact analysis algorithms. To investigate the relations between dependencies identified by programmers and impact analysis algorithms, we carried out a case study to examine the programmers' thinking during a program comprehension task and to find out what kind of impact analysis methods they are likely to apply. We learned that the qualifications and experience of the participants also affected the composition of the impact sets. Our main goal was to investigate programmers' strategies for understanding and utilizing relevant information and to find different tools to help them during impact analysis sessions.

Based on our analysis of the data collected during the study, we found that different programmers using different tools for different impact sets. Due to the different visualization technologies, the programmers also identified different dependencies. The majority of them neglected SEA relations and accepted all kinds of new dependencies. From a concrete list of dependencies (see BEFRIEND), they identified more dependencies. However, the new dependencies had a lower confidence level. Here we treated a dependency as a real dependency if the programmer voted to at least a 66% confidence level.

We found that SEA impact sets cover the programmers' impact sets the best. Most of their dependencies were SEA relations, and the developers and the students best recognized these kinds of dependencies. Without extra knowledge about impact analysis methods, they most easily recognized those methods which are executed after the method examined.

In contrast, PhD students thought in a systematic way. They covered best the dependencies identified by any impact analysis algorithms. They learned about these kinds of methods in courses and actively looked for them in the code.

References

1 **Orso A, Apiwattanapong T, Law J, Rothermel G, Harrold M J**, *An Empirical Comparison of Dynamic Impact Analysis Algorithms*, ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2004, 491–500.

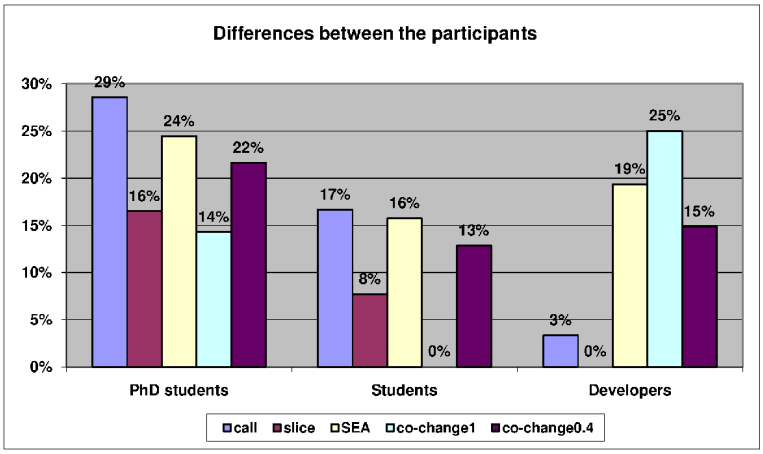

Fig. 4. Identified dependencies grouped according to participants' qualifications.

2 **Bible J, Rothermel G, Rosenblum D S**, *A comparative study of coarse- and fine-grained safe regression test-selection techniques*, ACM Trans. Softw. Eng. Methodol., **10**(2), (2001), 149–183, DOI 10.1145/367008.367015.

3 **Sridhara G, Hill E, Pollock L, Vijay-Shanker K**, *Identifying Word Relations in Software: a comparative study of semantic similarity tools*, Proc Int'l Conf. on Program Comprehension, IEEE, June, 2008, 123-132.

4 **Roy C K, Cordy J R**, *Scenario-Based Comparison of Clone Detection Techniques*, ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2008, 153–162, DOI http://dx.doi.org/10.1109/ICPC.2008.42, (to appear in print).

5 **de Alwis B, Murphy G C, Robillard M P**, *A Comparative Study of Three Program Exploration Tools*, ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2007, 103–112, DOI http://dx.doi.org/10.1109/ICPC.2007.6, (to appear in print).

6 **Ceccato M, Marin M, Mens K, Moonen L, Tonella P, Tourwe T**, *A Qualitative Comparison of Three Aspect Mining Techniques*, IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2005, 13–22, DOI http://dx.doi.org/10.1109/WPC.2005.2, (to appear in print).

7 **Lange C, Harry M. Sneed, Andreas Winter**, *Comparing Graph-Based Program Comprehension Tools to Relational Database-Based Tools*, International Conference on Program Comprehension, **0**, (2001), 0209, DOI http://doi.ieeecomputersociety.org/10.1109/WPC.2001.921732.

8 **Nicolas A**, *A Comparison of Graphs of Concept for Reverse Engineering*, IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2000, 231.

9 **Arnold R S, Bohner S A**, *Impact Analysis - Towards a Framework for Comparison*, ICSM '93: Proceedings of the Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 1993, 292–301.

10 **Breech B, Tegtmeyer N, Pollock L**, *A Comparison of Online and Dynamic Impact Analysis Algorithms*, CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, 2005, 143–152, DOI 10.1109/CSMR.2005.1, (to appear in print).

11 **Rajlich V, Gosavi P**, *Incremental Change in Object-Oriented Programming*, IEEE Software, **21**, (2004), 62-69, DOI http://doi.ieeecomputersociety.org/10.1109/MS.2004.17.

12 **Lindvall M, Sandahl K**, *How well do experienced software developers predict software change?*, J. Syst. Softw., **43**(1), (1998), 19–27, DOI 10.1016/S0164-1212(98)10019-5.

13 **von Knethen A, Grund M**, *QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces*, ICSM '03: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2003, 246.

14 *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers, 1990.

15 **Cohen J**, *A coefficient of agreement for nominal scales*, Psychological Bulletin, **20**, (1960), 37–46, `http://www.bibsonomy.org/bibtex/2495049be4ef603f5b67ce1dd7ecccdf8/chato`.

16 **Bohner S A**, *Impact analysis in the software change process: a year 2000 perspective*, ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 1996, 42–51.

17 **German M D, Hassan A E, Robles G**, *Change impact graphs: Determining the impact of prior codechanges*, Information and Software Technology, **51**(10), (2009), 1394 – 1408. Source Code Analysis and Manipulation, SCAM 2008.

18 **Marcus A, Sergeyev A, Rajlich V, Maletic J I**, *An Information Retrieval Approach to Concept Location in Source Code*, The 11th IEEE Working Conference on Reverse Engineering (WCRE'04), 2004.

19 **Rajlich V**, *Intensions are a key to program comprehension*, ICPC, 2009, 1-9, DOI 10.1109/ICPC.2009.5090022, (to appear in print).

20 **Hassan A E, Holt R C**, *Predicting Change Propagation in Software Systems*, ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2004, 284–293.

21 **Ryder B G**, *Constructing the Call Graph of a Program*, IEEE Trans. Softw. Eng., **5**(3), (1979), 216–226, DOI 10.1109/TSE.1979.234183.

22 **Horwitz S, Reps T, Binkley D**, *Interprocedural Slicing Using Dependence Graphs*, ACM Transactions on Programming Languages and Systems, **12**(1), (1990), 26–61.

23 *Indus project: Java program slicer and static analyses tools.*, `http://indus.projects.cis.ksu.edu/`.

24 **Zimmermann T, Weißgerber P, Diehl S, Zeller A**, *Mining Version Histories to Guide Software Changes*, IEEE Trans. Software Eng., **31**(6), (2005), 429-445, DOI 10.1109/TSE.2005.72.

25 **Kitchenham B A, Pfleeger S L, Pickard L M, Jones P W, Hoaglin D C, Emam K E, Rosenberg J**, *Preliminary guidelines for empirical research in software engineering*, IEEE Trans. Softw. Eng., **28**(8), (2002), 721–734, DOI 10.1109/TSE.2002.1027796.

26 **Pennington N**, *Comprehension strategies in programming*, Ablex Publishing Corp., Norwood, NJ, USA, 1987, 100–113.

27 **Basili V R, Selby R W, Hutchens D H**, *Experimentation in software engineering*, IEEE Trans. Softw. Eng., **12**(7), (1986), 733–743.

28 **Corritore C L, Wiedenbeck S**, *An exploratory study of program compre-*

hension strategies of procedural and object-oriented programmers, Int. J. Hum.-Comput. Stud., **54**(1), (2001), 1–23, DOI 10.1006/ijhc.2000.0423.

29 **Mosemann R, Wiedenbeck S**, *Navigation and Comprehension of Programs by Novice Programmers*, IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2001, 79.

30 **Ko A J, Myers B A, Coblenz M J, Aung H H**, *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks*, IEEE Trans. Softw. Eng., **32**(12), (2006), 971–987, DOI 10.1109/TSE.2006.116.

31 **Robillard M P, Coelho W, Murphy G C**, *How Effective Developers Investigate Source Code: An Exploratory Study*, IEEE Trans. Softw. Eng., **30**(12), (2004), 889–903, DOI 10.1109/TSE.2004.101.

32 **Tóth G, Hegedűs P, Jász J, Beszédes Á, Gyimóthy T**, *Comparison of Different Impact Analysis Methods and Programmer's Opinion – an Empirical Study*, The 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010), 2010.

33 **Weiser M**, *Program slicing*, ICSE '81: Proceedings of the 5th international conference on Software engineering, IEEE Press, Piscataway, NJ, USA, 1981, 439–449.

34 **Boehm B W**, *Software Engineering*, IEEE Transactions on Computers, **25**, (1976), 1226–1241, DOI 10.1109/TC.1976.1674590.

35 **Beszédes Á, Gergely T, Jász J, Tóth G, Gyimóthy T, Rajlich V**, *Computation of Static Execute After Relation with Applications to Software Maintenance*, Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07), IEEE Computer Society, Oct, 2007, 295-304.

36 **Jász J, Beszédes Á, Gyimóthy T, Rajlich V**, *Static Execute After/Before as a Replacement of Traditional Software Dependencies*, Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08), IEEE Computer Society, Oct, 2008, 137-146.

37 *JRipples tool for Incremental Change*, available at `http://jripples.sourceforge.net/`.

38 **Arnold R S, Bohner S A**, *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

39 **Weiser M**, *Program Slicing*, IEEE Trans. Software Eng., **10**(4), (1984), 352-357.

40 **Antoniol G, Rollo V F, Venturi G**, *Detecting groups of co-changing files in CVS repositories*, IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution, IEEE Computer Society, Washington, DC, USA, 2005, 23–32, DOI 10.1109/IWPSE.2005.11, (to appear in print).

41 **Buckner J, Buchta J, Petrenko M, Rajlich V**, *JRipples: A Tool for Program Comprehension during Incremental Change*, IWPC, 2005, 149-152, DOI 10.1109/WPC.2005.22, (to appear in print).

42 **Fülöp L, Hegedűs P, Ferenc R, Gyimóthy T**, *Towards a Benchmark for Evaluating Reverse Engineering Tools*, Tool Demonstrations of the 15th Working Conference on Reverse Engineering (WCRE 2008), Antwerpen, Belgium, Oct, 2008, DOI 10.1109/WCRE.2008.18, (to appear in print).

43 **Fülöp L, Hegedűs P, Ferenc R**, *BEFRIEND - a Benchmark for Evaluating Reverse Engineering Tools*, Per. Pol. Elec. Eng., **52**(3–4), (2008), 153–162, DOI 10.3311/pp.ee.2008-3-4.04.

44 **Fülöp L, Ferenc R, Gyimóthy T**, *Towards a Benchmark for Evaluating Design Pattern Miner Tools*, Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), IEEE Computer Society, Athens, Greece, Apr, 2008, DOI 10.1109/CSMR.2008.4493309, (to appear in print).

45 *The BEFRIEND homepage*, available at `http://www.inf.u-szeged.hu/befriend/`.