Abstract—Programmers tend to spend a lot of time debugging code. They check the erroneous phenomena, navigate the code, search the past bug fixes, and modify the code. If a sequence of these debug activities can be automated, programmers can use their time for more creative tasks. To address this problem, this paper proposes dcNavi (Debug Concern Navigator), a concern-oriented recommendation system for debugging. The dcNavi provides appropriate hints to programmers according to their debug concerns by using a repository containing not only program information but also past test results and their bug fixing history. We propose the notion of DCG (Debug Concern Graph), an extension of the Concern Graphs [10], [11] that helps programmers identify and reason about concerns in programming and maintenance tasks by abstracting the implementation details of a concern and makes explicit the relationships among concerns. A DCG appends a set of debugging information such as bug fixes and test results to a concern graph.

The remainder of this paper is structured as follows. In Section 2, we point out the issues on debugging. In Section 3, the concept of DCGs is illustrated. In Section 4, an overview of dcNavi is shown. In Section 5, the effectiveness of dcNavi is evaluated. In Section 6, related work is introduced. Concluding remarks are provided in Section 7.

I. INTRODUCTION

In many software development projects, programmers tend to spend a lot of time debugging code [5]. They check the erroneous phenomena, navigate the code that may include bugs, and search external useful resources such as API specifications and documents. It is effective if IDEs (Integrated Development Environments) provide programmers with advices summarized by these external resources. However, it is not clear which kind of information navigation should be provided to the programmers because these external resources provide only indirect hints for debugging and do not always resolve the bugs.

It is favorable if IDEs can provide the code snippets before and after similar bug fixes by searching past repositories because programmers can understand how to fix a bug by referring concrete examples. Especially, it is useful for novice programmers. To achieve this goal, we have to develop the following mechanisms because the code in the repositories should be associated to the debugging activities in order to provide effective hints: 1) a mechanism for collecting and archiving the debugging activities from IDEs; 2) a mechanism for associating the debugging activities to the program structures; and 3) a mechanism for retrieving the code snippets advising bug fixes from the repositories.

To deal with this challenge, this paper proposes dcNavi (Debug Concern Navigator), a concern-oriented recommendation system for debugging. The dcNavi provides appropriate hints to programmers according to their debug concerns by using a repository containing not only program information but also past test results and their bug fixing history. We propose the notion of DCG (Debug Concern Graph), an extension of the Concern Graphs [10], [11] that helps programmers identify and reason about concerns in programming and maintenance tasks by abstracting the implementation details of a concern and makes explicit the relationships among concerns. A DCG appends a set of debugging information such as bug fixes and test results to a concern graph.

The code below shows a unit test program for the readFile method. While the testReadFile method is successfully ter-

```java
01: public class Property {
02:     public String readFile (String pathname) {
03:         throws IOException {
04:             File file = new File(pathname);
05:             File file = new FileReader(file);
06:             BufferedReader br = new BufferedReader(fr);
07:             val = br.readLine();
08:             return val;
09:         }
10:     }
```
minated, *FileNotFoundException* is thrown in the case of the `testReadFileFileNotExist` method execution (line 13).

```java
01: public class PropertyTest extends TestCase {
02:   public void testReadFile() throws IOException {
03:     Property property = new Property();
04:     assertEquals("true", property.readFile("property.txt"));
05:   }
06:   public void testReadFileFileNotExist() throws IOException {
07:     Property property = new Property();
08:     // property2.txt does not exist !
09:     assertEquals(null, property.readFile("property2.txt"));
10:   }
11: }
```

The student begins to debug the program and is faced with the following three problems: 1) although the programmer guesses that he or she has to append the code for dealing with the case in which a file does not exist, the programmer does not know how to write the code avoiding *FileNotFoundException* because it is an unfamiliar exception for the programmer; 2) the programmer cannot be convinced of the correctness of the API usage (*BufferedReader*) because it is an unfamiliar library; and 3) the programmer does not know whether other test cases are needed.

Although it might not be difficult for professional programmers to predict the cause of a bug by referring to their experience, many beginners are faced with the various difficulties when they debug a program. For example, it is not easy for a beginner to use program exceptions or test failures as a hint for fixing a bug because they do not have enough experience and knowledge for programming languages, library usages, and source code reviews. Even if a programmer has a long career, he or she will be faced with the similar difficulties when the programmer works for unfamiliar new projects, languages, and frameworks.

**B. Our approach**

Many programmers tend to make the same kind of bugs concerning API usages, exception handling, and domain knowledge [4], [8], [9], [12]. It is effective to provide hints from past bug fix data.

This paper proposes *dcNavi*, a recommendation system for giving an answer to the programmer’s questions such as “how to handle this exception” and “how to use this API”. We call these questions *debug concerns*. The *dcNavi* provides hints for bug fixes by reusing debug experience and knowledge archived in the graph structures, DCGs, containing not only source code but also static analysis information, test results, and revision numbers before/after bug fixes. By using graph structures as knowledge bases, it becomes easy to navigate and search related information that may be hints for debugging. In *dcNavi*, a hint is provided in the form of code snippets showing the code modification before and after a bug fix because the correct code samples are effective for program understanding.

**III. DEBUG CONCERN GRAPH**

In this section, we explain the notion of DCG in detail.

**A. Graph structure**

Figure 1 illustrates two DCGs showing before and after debugging the `readFile` method from Section 2. The root
node `readFile debug` indicates the name of a debug activity consisting of sub-activities related to a test failure (diff-1) and a test success (diff-2). We consider test failure time and test success time as debug start point and debug end point, respectively.

The DCG consists of element nodes and edges. The nodes include not only program elements such as classes and methods, but also a variety of debugging activities annotated with applied bug fixes and test results. Each node is connected to other nodes by edges annotated with stereotypes such as `calls` (method call), `creates` (object instantiation) [some edges are omitted in Figure 1 for simplicity], `declare` (method declaration), and three kinds of relations: 1) `concerns` (relation between a debug activity and a class concerned by a programmer); 2) `test-result` (relation between a test class and its execution result); and 3) `bug-fix-pattern` (relation between a modified class and an applied bug fix pattern).

The bug fix patterns are the code modification catalogues for fixing a variety of bugs and can be considered hints for debugging. We adopted the bug fix patterns proposed in [9]. The patterns are well documented and consist of twenty-seven catalogues such as MC-DAP (Method Call with Different Actual Parameter Values), SQ-AROB (Addition or Removal of Method Calls in a Short Construct Body), and IF-CC (Change of If Condition Expression). The `dcNavi` detects the applied bug fix pattern by comparing the code before debugging with the code after the bug fix. In Figure 1, the SQ-AROB pattern is applied.

**B. Graph evolution**

A DCG is created and expanded automatically as a programmer debugs a program. First, a sub-activity-node is created when a test is executed. The structure of the `diff-1` node is fixed and the `diff-2` node is created at the time of test failure. If the test fails, a programmer begins to browse classes, methods, and fields that might include a bug. These browsed program elements are automatically added to a DCG. The result of the test execution is also added to the DCG. After the code modification is completed, the test is executed again. If the test succeeds, the bug fix pattern node is added to the DCG.

Our approach is integrated with TDD (Test-Driven Development) [2]. The TDD events such as test execution, fail, success can be good triggers for enriching concern graphs with debugging activities that are automatically associated to program elements such as classes and methods.

**C. Concern query**

It is a difficult problem to translate a programmer’s debug concern to a query for obtaining the code snippets showing the program modification before and after the bug fix because searching only keywords (e.g., `FileNotFoundException`) included in the debug concern matches with many code fragments that are not relevant to the bug fix. To deal with this problem, `dcNavi` searches a sub graph that matches the bug fix patterns and includes the specified keywords. The bug fix patterns show not only how to fix bugs but also the reasons of the bugs. The bug fix patterns play an important role in our recommendation system because many bug fixes can be categorized into the patterns, the advice can be generated from actual code snippets annotated with the bug reasons, and the recommendation mechanism is implemented by simple pattern matching.

**IV. dcNavi**

The `dcNavi` consists of three facilities including `DCG manager` for supporting graph construction and evolution, `DCG generator` for importing existing repositories, and `debug recommender`. Figure 2 illustrates the outline.

The `dcNavi` supports debugging in Java and is implemented as an Eclipse plug-in by using Mylyn and JUnit. Mylyn [7], [3] monitors programmer’s activities, creates a `task context` that focuses his or her workspace, and automatically links all relevant artifacts to the task context. JUnit is a unit testing framework for Java.

**A. DCG manager**

DCG manager creates and expands DCGs as follows: 1) program elements related to debug concerns are automatically captured and added to DCGs by using Mylyn’s task-focused interface (`concern` edges in DCGs are also appended automatically); 2) structural associations among program elements such as `declare`, `create`, and `call` are added to DCGs by analysing source code; 3) test results are added to DCGs by monitoring JUnit test executions; and 4) the applied bug fix patterns are added to DCGs by checking the `diff` before and after bug fixes determined by test failure and success events.

Programmers do not have to be aware of creating DCGs because the `dcNavi` automatically constructs DCGs based on Mylyn. Using Mylyn, the information only related to the debug activities is added to the DCGs. The static program analysis information, test results, and applied bug fix patterns are merged with Mylyn and Eclipse. Programmers can browse the debug information from the IDE because the `dcNavi` is tightly integrated to Eclipse.
B. DCG generator

Although our approach is effective for debugging, it needs a repository in which past debugging information is archived. This is not realistic because there are many repositories that do not take into account debugging activities.

To deal with this problem, we developed the DCG generator to convert a set of existing Subversion (SVN) repositories such as open source projects into DCGs. The DCG generator uses commit-log keywords such as bug, fix, and patch to define test success or failure, because a sub-activity node (e.g., \textit{diff-l}) in a DCG has to be created per each debugging process staring from a test failure and terminating by a test success. Although a DCG construction is based on TDD in \textit{dcNavi}, there are no data concerning TDD activities in most existing repositories. We have to predict and compensate this data. Moreover, we have to add program elements only concerning to the debug activities because a concern graph is automatically expanded by using the facility of Mylyn. The DCG generator links only program elements related to the bug fixes appearing in the commit-logs to the corresponding sub-activity node by analysing the program structures. That is, concerns edges are automatically added to DCGs.

C. Debug recommender

Figure 3 illustrates an example of a concern query “Search Related Modification” ofFileNotFoundException and its recommendation result. The \textit{dcNavi} explores a DCG from the testReadFileFileNotExist node (see Figure 1) and traverse connected nodes such as Property and readFile with comparing past bug-fixed graphs. The recommendation in Figure 3 shows that a file existence check should be inserted. In this case, the SQ-AROB pattern indicates that “Addition of Method Calls \texttt{[file.exists()]}” is needed for this bug fix.

This recommendation shows that programmers tend to forget to write this checking code.

The recommendation algorithm is as follows.

**Step1) Search a method to be modified:** The \textit{dcNavi} searches the test target method because our approach is based on TDD. This method needs to be modified. In this case, we have to find \texttt{readFile}, a method that throws the FileNotFoundException exception by tracing the \textit{call} edges in the DCG from the testReadFileFileNotExist node.

**Step2) Obtain recommendation candidates:** The \textit{dcNavi} obtains all the tests failing to handle the FileNotFoundException exception from the DCG. Next, the \textit{dcNavi} finds the test target methods whose \textit{after-modification} include the bug fix patterns that can be reached from the starting node such as FileNotFoundException. These test target methods are recommendation candidates. In the case of Figure 3, JUnitTestRunMonitor, a candidate of debug recommendation, includes the SQ-AROB pattern. Currently, the \textit{dcNavi} does not search the IF-CC pattern because it matches with many bug fixes and is not appropriate for obtaining essential bug causes.

**Step3) Recommend code snippets:** The \textit{dcNavi} recommends the code snippets ranked by the graph similarity metric:

$$\text{Similarity}(G_1, G_2) = \frac{\#\text{common_node}}{\left(\#G_1\_node + \#G_2\_node\right) / 2}. $$

This metric indicates the degree of the similarity between two graphs \(G_1\) and \(G_2\). \#\text{common_node} is the number of nodes commonly appearing in \(G_1\) and \(G_2\). This metric becomes high if the number of the nodes commonly shared between a current graph and a past bug-fixed graph is large.
In the case of Figure 3, `readFile` (before modification) and `JUnitTestRunMonitor` (before modification) correspond to G1 and G2, respectively. The code snippet of the `JUnitTestRunMonitor` (after modification) is useful for fixing the bug of the `readFile` method. As shown in Figure 3, it is easy for programmers to fix bugs by referring to concrete code snippets. Currently, the `dcNavi` supports several concern queries such as “Search Related API Usages”, “Search Test Cases”, and “Search Review Points”.

V. EVALUATION

In this section, we evaluate the effectiveness of our approach in terms of the recommendation quality.

A. Test data and Criteria

We generated DCGs by using nine open source repositories created in the Eclipse plug-in projects. We selected a set of repositories related to Mylyn because we predicted that there were similar bug trends in the same domain. First, we collected 4/5 revisions from each project and all revisions of other eight projects as the training data. Next, using this data, we checked how many recommendations were provided to the bugs contained in the remaining 1/5 revisions of each project. After that, we compared correct bug fix set with recommendations provided by `dcNavi`.

We consider that a recommendation is correct when the `dcNavi` recommends a bug fix pattern that is used in the real bug fix. More concretely, we define the criteria of correct recommendation as follows: 1) the bug fix pattern applied to the real debugging is the same as that of recommendation; and 2) all the method calls related to the bug fix pattern must be included in the method to be modified. We think that a past bug fix dealing with the same method calls can be a hint for debugging and the applied pattern indicates the reason of the bug. In this evaluation, we used the value 0.20 as the minimum graph similarity metric. Precision becomes high if we use a large value as the similarity metric. On the other hand, recall becomes low in this case. The average F-measure in all nine projects becomes most favorable when the value of the similarity metric is 0.20.

B. Evaluation results

Table I shows the results of the recommendations. The rough precision (the fraction of the correct recommendations among all recommendations) ranges from 15% to 35%. The rough recall (the fraction of the correct recommendations among all correct bug fixes) ranges from 20% to 50%. In case of `qcMylyn` (HP Quality Center Mylyn Connector), 174 (= 223 * 4/5) revisions and 10,035 revisions (other eight projects) are used as the training data. There are 54 methods including bugs and the average number of the provided recommendations per method (ANR) is 5.70. Precision and recall are 34.42% and 50.24%, respectively.

C. Discussion

Our criteria of the correctness is slightly rigorous because all the method calls related to the bug fix pattern must be included in the method to be modified (see condition 2). However, it is effective to reuse the code with the same structure in which only the called methods are different (e.g., code templates). Although `dcNavi` recommends this kind of code, the code is not treated as a correct answer in our evaluation because the condition 2 is not satisfied. On the other hand, there may be too many recommendations if the condition 2 is deleted. We have to relax the condition 2 in order to regard this type of code as a correct answer.

Someone might think that the percentage of correct recommendations provided by `dcNavi` is not high. One reason is the rigorous criteria used in this evaluation as mentioned above. However, it is not easy to define the correctness in essence because the usefulness of recommendations varies according to the programmer’s debug situation and skills.

To deal with this problem, we plan to introduce the recommendation options such as name matching and the value of minimum graph similarity metric. It would be better if programmers can specify the category of recommendations: API, exception handling, project-specific library, and so on. We think that the debug advice will be right to the point if the scope of the recommendation is explicitly specified. By introducing the recommendation options, we expect that we
can define the correctness suitable to each option. This is our future work.

VI. RELATED WORK

Debugging is one of the crucial issues in software engineering and many researchers have proposed a variety of support tools such as debuggers and static analysis tools. Although the information provided by these tools is effective, they cannot directly advise programmers on the bug fixes. Recently, several recommendation systems for debugging have been proposed to deal with this problem.

The Whyline [6] is a debugging tool that allows programmers to ask “Why did” and “Why didn’t” questions about the bugs. Programmers choose from a set of questions generated by static and dynamic analyses, and the tool provides answers of the bug causes. While Whyline uses static and dynamic analysis information of the target program, dcNavi uses not only program analysis information but also past bug fix information. We admit that fine-gained dynamic analysis information is effective for detecting the fault location and we plan to add the information to DCGs. If Whyline and dcNavi are integrated, we can provide more appropriate debug hints to the programmers.

The DebugAdvisor [1] is a search tool that allows programmers to express the context of the bugs and search through diverse data such as natural language text and core dumps. The DebugAdvisor allows programmers to search using a fat query, which could be kilobytes of structured and unstructured data. The dcNavi, which can be considered one of the search tools, explores structured DCGs focusing debugging. Although the goal of our approach is different from that of DebugAdvisor, we admit that large-scale searching is essential for the practical debug support.

The FixWizard [8] supports the tasks that identify the code peers existing in the program and recommend the similar fixes to its peers. Although FixWizard is similar to our approach, dcNavi focuses on the notion of Concern that can be related to a variety of debugging knowledge.

In dcNavi, the algorithm for searching the recommended methods is simple and general. It is effective that we can use the algorithm specific to the debug concern. The CAR-Miner [12] provides an approach for mining exception-handling rules as sequence association rules. These rules are specific to exception-handling.

The BugMem [4] provides a bug finding algorithm using bug fix memories consisting of project-specific bugs and fix knowledge. These bug fix memories use a learning process to extract project-specific bugs.

VII. CONCLUSIONS

This paper proposed a new concept, debug recommendation based on Concern Graphs, for providing the appropriate hints to programmers according to their debug concerns. Adopting our approach, programming, testing, and debugging can be systematically integrated based on the Concern Graphs. Although our approach is effective as demonstrated in Section 5, we have to improve the quality of the recommendation. As the first step, we plan to integrate dcNavi with runtime verification and dynamic analysis.

REFERENCES