Improving the Teaching of Software Design with Automated Modelling of Syntactic Dependencies

Kevin Steppe, Sally Chin, Wong Wai Tuck
School of Information Systems
Singapore Management University
kevinsteppe@smu.edu.sg
sally.chin.2015, wtwong.2015 @sis.smu.edu.sg

***Abstract*— We present the use of a new IDE plugin for introducing students to the analysis of software design. Without a concrete method to evaluate their ideas, designing for modifiability was a challenging topic for our students. Prior work showed that students can quickly learn about dependency graphs and use them to make design decisions. However, students frequently made mistakes creating the graphs and identifying ripple effects. We developed a tool that automatically generates dependency graphs from code. The plugin allows users to select seed modifications and then highlights dependent modules. The tool removed the common mistakes from the process and enabled us to teach design to students with less experience. In this paper, we present our findings teaching workshops for second-year undergraduates using the tool. The students were able to use the tool to analyze and compare designs. Students indicated they are likely to continue to use the technique.**

Keywords*— *Software Engineering Education; Design; Dependency Graph; Design Learning

I. Introduction

Modifiability is critical for any software system – to ease initial development and future changes. Effective modularity and minimization of ripple effects have long been recognized as key aspects for a modifiable design [1,2,3,4]. Much research work has been done in software design to promote modifiability – polymorphism, patterns [5], aspects, messaging middleware, web services, and more. However, modularity and comparing designs are typically taught to software engineering students as informal principles and heuristics. Such techniques require considerable experience to apply well. When separating concerns, which functions should be separated and which encapsulated? Does an adaptor, which introduces more calls, promote loose coupling? It can be difficult to select between alternative designs, since there is no definite answer of whether one design is better than another, and few techniques or means exist for making such decisions. Students rarely have the opportunity to evolve homework assignments and thus develop limited experience and intuition regarding modifiability [14].

A study by Rupakheti and Chenoweth found that undergraduate students prefer "canned homework and tutorials" over open-ended problems [15]. In this context where teaching software design and modifiability is concerned, it would be preferable if the concepts taught were more concrete and could be applied through direct rules instead of subjective experience.

The Dependency Graph Method (DGM) was developed to enable students to analyze software designs [25]. The DGM was constructed to be consistent with commonly practiced design patterns [5], to allow comparative analysis of different designs, and be objective enough to be applied correctly by novices. As our students frequently have trouble applying subjective judgments about 'uses', encapsulation vs. separation and other principles, an objective evaluation technique is needed.

However, we found that students often made errors in identifying dependencies and correctly identifying the dependent set of a given module [25]. As mentioned in that paper, those errors could be removed through tooling. Further, tooling can improve code and design understanding. In a study by Szabo, it was found that students' approach to code understanding can be significantly improved through the use of tools [24]. Similarly, Cai and others have used tools to aid students' understanding of software design [11,12].

In this paper we present a tool to automate the creation of dependency graphs and analyze the impact of this tool on software design education. *D–Grapher* is a plugin for IntelliJ that produces dependency graphs from code. The tool also allows users to select modules of interest and highlights the relevant dependencies. We used the tool to teach design analysis to 60 undergraduate information systems students. We conducted a one-day workshop including motivation, a discussion of the Dependency Graph Method, training on *D–Grapher*, followed by examples and exercises. We ended the workshop with a short test covering the same topics as the evaluation in [25] but with access to the *D–Grapher* tool. As expected, the tool completely eliminated errors for tasks the tool automates. For design decision tasks we also see an improvement.

D–Grapher supports most Java programming structures. Currently, annotations used at precompile time are not covered. Java9 features have not been tested.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 explains the dependency graph method and the *D–Grapher* tool. Section 4 discusses the method of the study and questions used for evaluation. Section 5 discusses the results and Section 6 concludes.

2377-570X/17 $31.00 © 2017 IEEE
DOI 10.1109/CSEET.2017.31

II. RELATED WORK

The DGM technique extends from several earlier techniques in analyzing dependencies, including Parnas's early concept of 'uses' structures [6], Jackson's analysis of assumptions [7], indirect coupling [8], and design structure matrices (DSM) [3,9]. Of these, our dependency graph technique is most similar to DSMs. This technique uses modules (classes or packages) as the nodes of a graph. There are two categories of dependencies: 'semantic' for data and functional dependencies and 'syntactic' for code level references. Semantic dependencies are considered to be transitive while syntactic dependencies are not. Also different from DSMs is that dependencies are explicitly directional. This structure allows us to make objective and comparative evaluations of designs.

This is similar to change impact analysis, where the estimated impact set (hereafter called the Dependent Set) is derived from impact analysis [19]. Impact analysis has shown to be useful if automated [20]. Impact analysis correlates with the change effort [23] and hence a forward looking estimated impact serves as a useful basis for design evaluation.

Existing impact analysis tools include evolutionary coupling techniques [18] which results in incomplete dependency generation [20] (such as EvoLens [17]), and call graph dependency resolution (such as JRipples [19]). The latter suffers from gross overestimation. For example in one visualization tool, the evaluators of the tool remarked that the tool often produced dependent sets that were so large that it was disorienting [16]. Ratzinger et al. showed that these visualization tools can help manual change impact analysis [17]. We have adapted these solutions to produce a visualization tool that improves on evolutionary coupling by using call graph dependency, which has been shown by Tóth et al. to be nearly as effective as static execute after (SEA) for change impact analysis [21]. We use a non-transitive propagation strategy, which better explains design patterns, such as adapter and factory, than fully-transitive propagation strategies, such as network analysis [22].

This study is also similar to the work on DSMs in education [11,12]. We are also working to bring improved analysis of design modularity and modifiability into software education. The DSM papers focus on conformance of implementation to an instructor specified design and uses tool support to assess that conformance. The authors also examine the causes for non-conformance and how much instructor support in reviewing the tool's output is needed for students to identify that non-conformance. Our study uses tooling to relate implementation to design level dependencies and focuses teaching on design questions – which alternative design will best handle a given change, and how can expected variations be protected.

III. DEPENDENCY GRAPH METHOD

Steppe introduced the dependency graph method as a means of comparing two competing designs [25]. While the dependency graph model can apply at any level of granularity, for the purposes of the tool and this paper, we look at syntactic dependencies at the class level. The dependency graph is a concise graph language sufficient to explain a range of existing best practices, compare different designs for a software system, and provide guidance for improving those designs. The graph model does not aim to provide new solutions to any particular design problem. Rather, it provides a 'language' and statements about structures expressed in that language thus allowing analysis of a wide range of designs.

The graph model describes systems as a composition of behaviors – functions within the system – the modules which implement behaviors, data exchanged by those modules, and interfaces to those modules. A module is defined as an atomic, independently editable piece of system implementation. This is a piece of system code – either procedural or declarative – which is separate from other modules in its editable representation. We typically define a class to be a module and use that definition in *D–Grapher*.

The model has two kinds of dependencies to show relations between the elements. Behaviors can depend on other behaviors through 'semantic' dependencies. These represent one functionality delegating responsibilities to other behaviors – similar to Jackson's assumptions [7] and Yang's indirect dependency [8]. The second category of dependencies is syntactic dependencies. These arise from the syntax of the implementation. These include a module implementing an interface, having a reference to another module, or by reference to a data type (in OO systems, a type is often a module).

Given the limits of the semantic dependencies, the designer's goal is to find a structure which works while maximizing the modifiability of the system implementation. To make implementations changeable independently, they must be divided into independent modules. However, any structural dependencies will inhibit this independence. In this sense, all structural dependencies restrict modifiability. The work of the designer is then to find a balance between decomposition and dependencies, and to structure those dependencies to optimize modifiability for modules likely to be changed. A more thorough description of the model is provided in [25].

A. Formal Definitions

Let $\boldsymbol{C}$ be the set of all modules (classes here) of a program. We define seed behavioral changes as the set of changes, $\boldsymbol{B}$. Each seed behavioral change $\boldsymbol{b} \in \boldsymbol{B}$ represents a request to change or add functionality of the program. We define a **seed set** $\boldsymbol{C_s(b)}$, which is the minimal set of classes whose functionality must be changed/added in the change request

For each of these **seed modules** $\boldsymbol{c_s} \in \boldsymbol{C_s(b)}$, let the dependent module $\boldsymbol{c_d}$ be a module that references $\boldsymbol{c_s}$ in the source code (i.e., $\boldsymbol{c_d}$ has a directed non-transitive **syntactic dependency** on $\boldsymbol{c_s}$). Examples include statements in $\boldsymbol{c_d}$ that make method calls or access the fields in $\boldsymbol{c_s}$, or $\boldsymbol{c_d}$ extending class $\boldsymbol{c_s}$. We use $\boldsymbol{Dep(c_s)}$ to denote the set of modules dependent on seed module $\boldsymbol{c_s}$, where $\boldsymbol{c_d} \in \boldsymbol{Dep(c_s)}$. We define the **dependent set** to be the set of all dependent modules of a given a seed set $\boldsymbol{C_s(b)}$ from a behavioral change $\boldsymbol{b}$:

$$\mathrm{Dep}(\mathrm{b}) \equiv \bigcup_{c_s \in C_s(\boldsymbol{b})} \mathrm{Dep}(c_s), \boldsymbol{b} \in \boldsymbol{B}$$

A ripple is a modification implemented on a module that is not a seed module:

$$c_r \in \text{Dep(b)}, \textit{where } c_r \notin C_s(b), b \in B$$

The **ripple set** is

$$R(b) \equiv \{\forall c_r\} \textit{ for } b \in B$$

Software engineers wish to minimize effort in modification: Minimize $|R(b)|$ for a given $b \in B$.

Since R(b) cannot be found directly before change, we hypothesize that |Dep(b)| is a useful estimate, and hence comparing |Dep(b)| would help us decide which design is better given a behavioral change b .

B. Interpreting Dependency Graphs

We have defined semantic dependencies such that they are inherent in the structuring of the solution – no rearrangement of code into different chunks will remove them. Thus when we teach the Dependency Graph Method to undergraduates, we focus on syntactic dependencies. We take syntactic dependencies to be non-transitive. The intuition can be seen by considering a client module's requests being passed through an adapter module that forwards them to a service. Should the syntax of the service be changed (a change in address, API naming, etc.) the adapter is modified but the client module does not need to be modified; that is the purpose of the adapter. Preliminary studies suggest that the size of the set of dependent modules, Dep(b), which are derived from the non-transitive directed syntactic dependencies on the seed modules, is a better estimate of the ripple set than the fully transitive propagation [28].

Given the graph structure, directed and non-transitive syntactic dependencies, we can assign modifiability properties to pairs of nodes based on whether or not there is a dependency between them. The *Changeable* property derives from the idea of protected variation – other modules are protected from variation in the *changeable* module. *Changeable* is defined as: "The implementation module X is labeled *changeable* respective to module Y if and only if: 1) X and Y are separate modules and 2) there are no direct syntax dependencies from module Y to module X. Part 1 of the definition ensures that separation of concerns is identified – failure to separate concerns results in modifying larger and more complex modules.

From this definition we can then define two sets for each seed module. All modules with a direct syntactic dependency on the seed module are in the dependent set. All other modules, with no direct syntactic dependency on the seed module are protected from the change. Thus in comparing alternative designs, we prefer the smaller dependent set for each expected change.

IV. D-GRAPHER

We introduce ***D-Grapher***, a plugin for IntelliJ. It extracts the Abstract Syntax Tree using the Program Structure Interface in IntelliJ. The tool parses Java programs to produce a dependency graph. From the dependency graph, the developer is able select seed modules for a requirement change he is considering, and the tool will derive the dependent set based on his selection and that will be produced as a graph for the user, as shown in Figure 1.

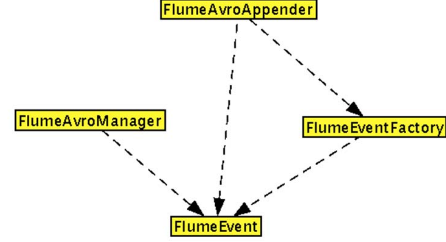

Fig. 1. Non-transitive dependencies of FlumeEvent, FlumeAvroAppender and FlumeEventFactory, from an actual change made in the open source Java project log4j. The arrows show direction of dependencies, so for example, FlumeAvroAppender depends on FlumeEventFactory.

Internally, IntelliJ parses the Java project and stores it in a data structure that represents the abstract syntax tree, and this data structure can be accessed using the Program Structure Interface (PSI). In this section, we will talk about the data structures used for representing the dependencies and the algorithms used to parse the abstract syntax tree.

A. Parsing the Abstract Syntax Tree

When the user requests the D–Grapher Tool Window, we parse the AST provided by IntelliJ. We start by retrieving all top level packages, which makes it possible to show multiple independent applications in a single window. This can be useful for making comparisons between alternative designs. From the top level packages, we collect all classes and sub-packages until all classes are found. Each class is added to the graph as a node.

Each class represented as an object of type *PsiClass*. We recursively parse children of the class using a depth first search. Each child (a code element within the class) is of type *PsiElement* and the method of parsing is different depending on the construct the actualized type represents. Each subclass of *PsiElement* represents a specific Java programming construct, and is therefore parsed differently in order to derive the classes that are referenced in each *PsiElement*. Once the referenced classes are found the dependency is added to the graph as a directed edge between the referencing class and referenced class.

D–Grapher supports most Java programming structures. Currently, annotations used at precompile time are not covered. Java9 features have not been tested.

B. Visualizing the Graph

Previously learned concepts and different graph notations can interfere with the learning of a new language [10]. Our students have prior knowledge of UML class diagrams, whose notation differs from the published DGM notation. In DGM notation, syntax dependencies are represented by solid arrows. For mapping of semantic dependencies, dashed arrows are used.

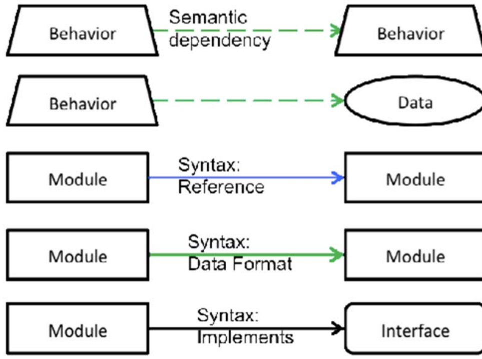

Fig. 2. Notation for Dependency Graph Method from Steppe [25].

However, in class diagram notation, dashed arrows represent the 'uses' relationship, which conceptually is closer to DGM's syntax dependency.

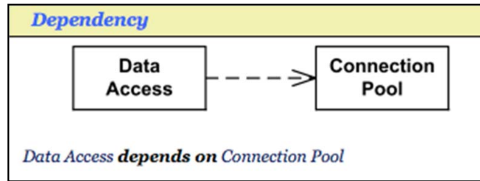

Fig. 3. UML class diagram's dependency notation.

We felt that the change in notation would cause confusion for students. Additionally, *D–Grapher* only displays syntax dependencies. For these reasons we decided to have *D–Grapher* follow the UML notation, with dashed arrows representing syntax dependencies (which are a more rigorously defined version of the class diagram "uses").

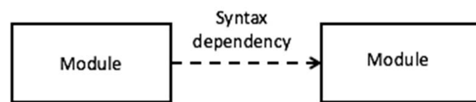

Fig. 4. D-Grapher's Revised dependency notation.

For the graph lay out we use Prefuse. Prefuse uses a spring and gravity model to lay out the graph. This provides a clean lay out for small graphs. For large applications, and when the design is poorly modularized, overlaps of dependencies are inevitable. We also provide the option to output a matrix representation of dependencies. Because Prefuse is based off of Java Swing, we have native integration with the UI elements in IntelliJ (i.e., the JPanel objects in the Tool Windows).

Further we use a signaling technique to distinguish between seed modules and dependent modules [27]. Signaling is a technique recommended to be used in environments to signal semantically important information, and it has been found to improve comprehension of the signaled material [26]. We highlight selected seed modules in yellow and dependent modules in green. This serves to ensure that the graph is read correctly (a problem encountered without the tool) and to direct the user's attention as the graph gets more complex.

Fig. 5. D-Grapher's display of a Seed change, DependentModule class and a class NotDependent. NotDependent has a white background, while clicking on the SeedModule turns it yellow and the DependentModule turns to green.

In Figure 6 we show the dependency graph for a small part of a potential design of a vending machine. The SelectionPanel module displays the sodas available and calls the SodaDispenser when a soda is selected. The SodaDispenser emits cans and informs Soda to change the inventory. We can see that the SodaDispenser makes calls to the Soda module and hence has a syntactic dependency on it. SodaDispenser is called by the SelectionPanel and thus is depended on. We can immediately see that SelectionPanel is "changeable" meaning that modifications to SelectionPanel, including API changes, have no impact on the other modules. With the tool, the SelectionPanel is marked as a seed, highlighted in yellow and the user will immediately notice that there are no green dependent modules.

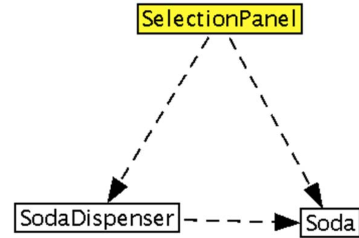

Fig. 6. Dependency graph for part of a design of a vending machine.

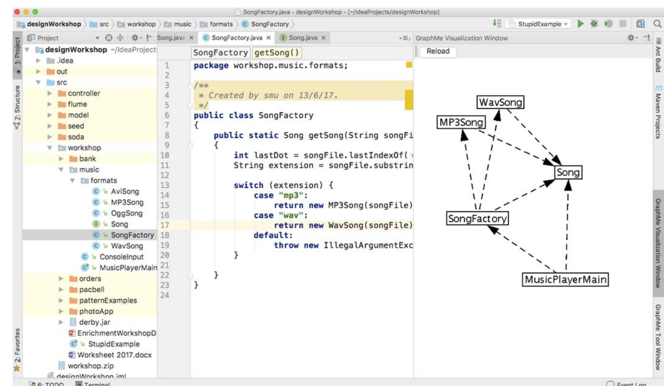

Fig. 7. Example of the *D–Grapher* interface in IntelliJ

C. *Example use of D–Grapher*

As an example use of the Dependency Graph Method with *D–Grapher*, we present an evaluation of two candidate designs for a small project. Our sample project takes orders from the user and communicates those orders to the AccountDepartment and CustomerRecords modules. In the "Service" design (see Figure 8), the User module sends an Order to the OrderTaker. The OrderTaker sends order information to CustomerRecords

and AccountDepartment. Those modules have an API requiring only the data elements they need from the Order.

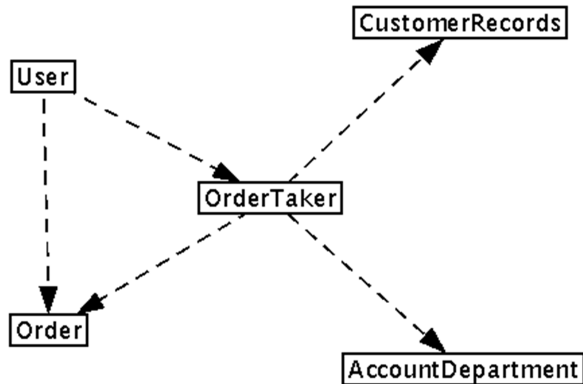

Fig. 8. Dependency Graph for Service Design

In the observer design (see Figure 9), the OrderTaker is an Observable and CustomerRecords and AccountDepartment are Observers. Note that in our example implementation, the dependency from CustomerRecords and AccountDepartment to the Observer and Observable classes are not shown as *D–Grapher* omits all JDK classes. In this design, whenever a user submits an Order, the OrderTaker sends the Order to all registered listeners without needing to know which modules are registered as listeners.

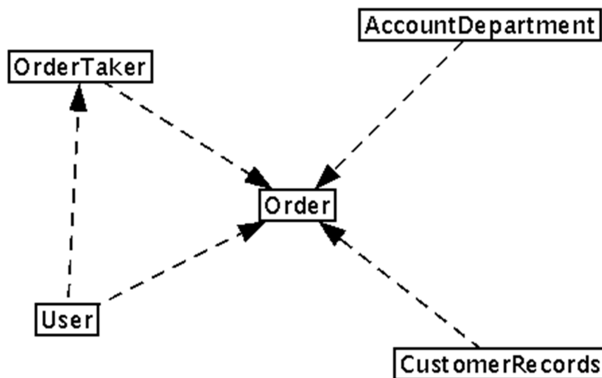

Fig. 9. Dependency Graph for Observer Design

Let us assume that the business wants to collect more information from the Order into CustomerRecords. The *D–Grapher* using developer selects CustomerRecords as the seed change (turning it yellow). In the "Service" design, the OrderTaker is highlighted in green, showing the developer that it is likely to need modification as well. However, when the developer looks at the "Observer" design, she sees that there are no dependent modules, making it the preferred design for this change.

Of course we can easily imagine a change where the "Service" design is preferable. If the developer selects the Order class as a likely change, she will see that in the "Service" design the User class and OrderTaker are likely to need adjustment (entering extra fields and reading them). While in the "Observer" design, all classes need modification as all interact with the Order class directly.

V. Study Methodology

We planned to test whether students would be able to learn and apply the Dependency Graph Method better with the tool than without one. Our hypothesis was that the tool would eliminate errors for tasks that the tool could automate, enable less experienced students to perform the same tasks as experienced students, and to improve the outcomes for decision making tasks.

For this study we had sixty student volunteers sit for a one-day workshop on the technique. Most had just completed the first year of an undergraduate information systems degree – 15% had completed the second year. All had completed at least two semester-long courses on Java programming. We note that these participants had considerably less experience than the participants in the prior study [25]. Based on past class grades, two students (3%) in this study were from the top quartile of their cohort, thirty-two students (53%) were from the second quartile, twenty-four students (40%) were from the third quartile and two students (3%) were from the bottom quartile.

All of the first-year students participants – 85% – had no prior experience with IDEs. The second year students – 15% – had used NetBeans for one semester. None of the second year students had prior experience with IntelliJ. Hence, we added half a day to the workshop to familiarize them with the basics of IntelliJ.

The workshop consisted of a lecture portion, which included motivation for the technique and covered the dependency graph method presented above. We presented examples of creating a graph based on code or UML diagrams. We also presented examples of dependency and ripple sets for a simple graph of domain object, data access object, and persistent storage. We discussed example cases using the technique to decide between alternative designs and using the technique to improve designs.

After explaining the method manually, we gave the students the *D–Grapher* plugin. We illustrated how to use the tool to produce dependency graphs, select change seed modules and identify dependent modules. As the tool is mostly automated with few actions required from the user, we found that students learn its use within an hour and are able to use the tool and method to make design decisions within the one-day workshop.

We then gave the students a set of scenario based exercises. The scenarios used are based off class practices and common mistakes in software design that novices may make, as we want to make these scenarios as relevant and realistic as possible from an undergraduate student's point of view. The scenarios give students practice reviewing designs with *D–Grapher*, comparing designs, and implementing changes on the selected design. For example, our scenarios include a simplified e-commerce site where the shopping may be replaced, a photo display application where the gallery API may change, and a bank transaction processor whose data source will change.

At the end of the workshop we had the students attempt a test. We used this test to assess whether the students were able

to correctly apply the technique and thus whether it is usable by novices. The students were allowed to use the *D–Grapher* tool during the test. We had five categories of questions to assess five tasks the students should be able to complete.

The first category of questions requires the student to take a few code samples and produce a dependency graph showing the syntactic dependencies in the code. In the second category of questions, the students were given sequence diagrams and asked to produce a dependency graph to match each diagram. These test their ability to relate other artifacts to dependency diagrams.

In the third category, we gave the students a dependency diagram plus a change scenario, including which modules are the seed of the change and asked them to determine which other modules might need to be modified due to ripple effects. This tests the students' understanding of how to use the graphs for change analysis.

In the fourth category we gave the students dependency diagrams for two alternative designs plus a change scenario and asked them to determine which of the alternatives would respond to the change better. We use these questions to test if the method enables the students to make comparisons as intended by the method.

Lastly, we presented the students with questions giving a candidate design plus change scenario and asked them to modify the design to make the change easier to accommodate. The intention was to see if the students understand how a good design protects the system from variation and if they can use that understanding to turn a poor design into a better one.

We expected that use of the *D–Grapher* tool would result in significant improvements for categories one and three (producing a graph from code and identifying dependent modules). After the assessment students were asked to fill out a short survey of their experience with the method and tool.

VI. RESULTS

In Table 1 we show the percentage of correct answers by category of question for this study. Table 2 shows the prior results without tool support. We discuss these results in more detail below.

TABLE I. PERCENTAGE OF CORRECT ANSWERS BY QUESTION CATEGORY WITH TOOL SUPPORT

Code to Graph	Sequence Diagram to Graph	Identify Ripples	Choose Alternative	Improve Design
100%	85%	100%	90%	80%

TABLE II. PERCENTAGE OF CORRECT ANSWERS BY QUESTION CATEGORY WITHOUT TOOL SUPPORT[25]

Code to Graph	Sequence Diagram to Graph	Identify Ripples	Choose Alternative	Improve Design
62%	92%	53%	88%	85%

As expected, the "code to graph" tasks were executed perfectly with tool support. This is no surprise as the tool generates the graph for the user. Similarly, "identify ripples" questions were answered perfectly when students had tool support. With the tool these questions only required the student to select a change seed module (which was given) within the graph window and read the highlighted dependent modules. These two results show that we achieved our goal to make these tasks easy through tooling.

In the past a common error was misinterpreting the arrow directionality. We have chosen for arrows to follow the direction of dependence. This means that ripple effects go the opposite direction of the arrows – modifications to B could ripple to A. The previous study showed that some students interpret the arrow direction inconsistently and thus expect different effects than the graph indicates. Here the tool's highlighting of seed and dependent modules significantly improved the student's understanding of dependency graphs.

The accuracy of choosing alternatives stayed about the same and the correctness of improving given designs dropped slightly. This shows that even after just one year of programming experience, most students are able to use the method and tool to analyze software designs. The lack of improvement from the with-tool group might be evidence of catching-up (they would have done much worse without the tool as they had less experience than the without-tool group). Or it could be that our questions, which were mostly based on usage of patterns, were too simple.

The accuracy of translations from sequence diagrams to graphs dropped in this study. We suspect the drop is due to a combination of less experienced participants and less time spent in training this particular task. However, this also raises the concern that our participants were able to follow the rules of the method without having a strong understanding of what dependencies represent at the code level. The ability of students to see the linkage between code structures and dependencies inhibiting modifiability is explored in [11].

Table 3 shows the average response (out of 10) to our survey questions. We find that the students generally have a favorable impression of the usefulness of the method and tool and report that they are likely to use them for future projects. We note that the coloring of modules was less helpful than we originally expected.

TABLE III. SURVEY OF PARTICIPANTS' IMPRESSION OF THE METHOD AND TOOL

Student's ratings	Average Score (Max score: 10)
Usefulness of the DGM methodology	8.37
Usefulness of D-Grapher	8.1
The coloring of modules helped to analyze the graph	7.81
Likelihood of using the tool for future projects	8.4

A. Limitations

The *D–Grapher* tool is limited by the identification of seed modules. Developers have to manually select the seed set for a given requirement change, and this often requires some experience. If students fail to select the correct seed modules, the highlighted dependent set will not be a sensible estimate of the actual impact.

We also find that students had more difficulty as the dependency graphs got more complicated. In a study with Genting, we found that even for a large program (over one million lines of code), the developers were able to limit cycles and keep the structure understandable. However, the graphs became very large for some changes and in some areas became very complicated. Graph lay out is a difficult problem and it is likely impossible to produce a clean (no overlapping nodes or edges) graph for more complex systems. To combat this, we plan for *D–Grapher* to provide ways to hide classes from the graph window.

This paper compares the performance of a with-tool group to a without-tool group. The without-tool study was done with an earlier cohort of students. While the courses both groups took were largely unchanged, there may have been changes which impacted their performance in this study. Further, the without-tool group had more experience, having been drawn from second and third year students. This may be why there is little difference in success choosing between designs.

VII. Conclusion

We reported using tool support to improve the use and teaching of the Dependency Graph Method [25]. We taught the method and the tooling to a group of mostly first-year undergraduate students in a one-day workshop. We explicitly tested the students' ability to apply this technique to analyzing the impact of changes on given designs and to choose between alternative design options. Our tests find that most students are able to make good design choices after the workshop. As expected, we find that the number of errors in creating and interpreting dependency graphs is reduced to zero through use of the tool support. Based on the survey, students found both the dependency graph method and *D–Grapher* useful for evaluating software.

Acknowledgment

The authors would like to thank Eric Nyberg from Carnegie Mellon University, for extensive advice in formulating both the technique and the study. The authors would like to thank Richard Davis of Act8Design for advice on user-interface design. The authors would also like to thank the reviewers for their valuable advance and corrections.

References

[1] D. Parnas, "On the criteria to be used in decomposing systems into modules," Communications of the ACM, 5(12). 1972.

[2] Yassine, A., et al., "Information hiding in product development: the design churn effect." Research in Engineering Design, vol 14, pp. 17, 2003.

[3] C.Y. Baldwin and K.B. Clark, Design Rules, Vol 1: The Power of Modularity. MIT Press, 2000.

[4] F. Wilkie and B. Kitchenham, "Coupling measures and change ripples in C++ application software," The Journal of Systems and Software, vol. 52 pp. 8, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2002.

[6] D. Parnas, "Designing software for ease of extension and contraction." IEEE Transactions of Software Eng., 5(2), 1979.

[7] D. Jackson, "Module dependences in software design", in Monterey Workshop on Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy, 2002

[8] H. Yang, E. Tempero, and R. Berrigan, "Detecting Indirect Coupling", Australian Software Engineering Conference, 2005

[9] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. "Analyzing the evolution of large software systems using design structure matrices and design rule theory." In Proc. 7th WICSA, pages 83-82, 2008.

[10] J. Bonar and E. Soloway, "Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers." Studying the Novice Programmer, Lawrence Erlbaum Associates: 325-353, 1989.

[11] Y. Cai, R. Kazman, C. Jaspan and J. Aldrich. "Introducing Tool-Supported Architecture Review into Software Design Education." In Proc. of the 26th IEEE Conference on Software Engineering Education and Training (CSEE&T), 2013.

[12] Y. Cai, D. Iannuzii and S. Wong, "Leveraging Design Structure Matrices in Software Design Education". In Proc. of the 24th IEEE Conference on Software Engineering Education and Training. 2011.

[13] P. Bhatt, G. Shroff, A. Misra, "Dynamics of software maintenance." ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, 2004

[14] Bengtsson, N. Lassing, Bosch and H. Vliet, "Analysing software architectures for modifiability", Vrije Universiteit, 2000.

[15] R. Rupakheti and Chenoweth. "Teaching software architecture to undergraduate students: An experience report." IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2, pp. 445 – 454, 2015.

[16] G. Pirklbauer, C. Fasching and W. Kurschl, "Improving change impact analysis with a tight integrated process and tool," Seventh International Conference on Information Technology: New Generations (ITNG), 956-961, 2010.

[17] J. Ratzinger, M. Fischer and H. Gall, "EvoLens: Lens-view visualizations of evolution data," Eighth International Workshop on Principles of Software Evolution, 103-112, 2005.

[18] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering, 31*(6), 429-445, 2005.

[19] S. Bohner and R. Arnold, Software Change Impact Analysis, IEEE Computer Society Press. 1996.

[20] T. Wetzlmaier and R. Ramler, "Improving manual change impact analysis with tool sSupport: A study in an industrial project," Lecture Notes in Business Information Processing Software Quality. Software and Systems Quality in Distributed and Mobile Environments, pp 47-66, 2015.

[21] G. Tóth, P. Hegedűs, A. Beszédes, T. Gyimóthy, and J. Jász, "Comparison of different impact analysis methods and programmer's opinion," Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, 2010.

[22] R. Wang, R. Huang, and B. Qu, "Network-Based Analysis of Software Change Propagation" The Scientific World Journal, 2014.

[23] A. Mockus, L. G. Votta. "Identifying reasons for software changes using historic databases". In Proceedings of the International Conference on Software Maintenance, 2000.

[24] Szabo, C., "Novice code understanding strategies during a software maintenance assignment", 37th IEEE International Conference on Software Engineering, Vol. 2, pp. 276 – 284, 2015.

[25] K. Steppe, "Teaching Analysis of Software Designs Using Dependency Graphs", 27th Conference on Software Engineering Education and Training (CSEE&T), pp 65-73, 2014.

[26] E.M. Gellenbeck and C.R. Cook, "Does signaling help professional programmers read and understand computer programs?" Empirical Studies of Programming: Fourth Workshop, 82-98. 1991

[27] J.F. Pane and B.A. Myers, "Usability issues in the design of novice programming systems," Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132, 85 pages. 1996.

[28] K. Steppe "A Dependency Graph Method for Analyzing Software Modifiability" Ph.D. Dissertation, Singapore Management University, 2015.