4-2014

# Teaching analysis of software designs using dependency graph

Kevin STEPPE

*Singapore Management University*, kevinsteppe@smu.edu.sg

## Citation

STEPPE, Kevin. Teaching analysis of software designs using dependency graph. (2014). *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T): Proceedings: April 23-25, Klagenfurt, Austria*. 65-73.

**Available at:** https://ink.library.smu.edu.sg/sis_research/7031

# Teaching Analysis of Software Designs using Dependency Graphs

Kevin Steppe
*School of Information Systems, Singapore Management University*
*kevinsteppe@smu.edu.sg*

## *Abstract*

*We present the use of a new type of dependency graph to aid students in analyzing the modifiability of software designs. Though a variety of software design concepts, such as information hiding, separation of concerns and patterns are taught to undergraduate students, they often have difficulty applying these concepts to the analysis of designs and particularly to comparing designs, perhaps due to the subjective nature of these concepts. Our new technique complements design structure matrix and 'uses' techniques to handle asymmetric dependency impacts and provide a deterministic approach to comparing alternative designs. A major goal of this technique was for students to be able to quickly learn about dependencies and use them to make design decisions. In this paper we present findings from a study with thirty third- and fourth-year undergraduates indicating that most were able to use the technique to analyze and compare designs after a single short workshop and indicate that they are likely to continue use the technique in the future.*

## 1. Introduction

Modifiability is critical for any software system –to ease both 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, guidance on 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 or does an adaptor, which introduces more calls, promote loose coupling? Students rarely have the opportunity to evolve homework assignments and thus develop limited experience and intuition regarding modifiability. Given these imprecise and sometimes subjective techniques it is difficult to assess students' ability to evaluate designs.

To improve this situation, we are testing a new technique based on dependency graphs to enable students to analyze and compare designs. In developing this technique we had 3 major goals: The technique must produce design analysis and advice consistent with practiced design patterns [5], it must aid in the comparative analysis of different designs, and it must be sufficiently precise and objective to be applied correctly by novices. The last follows our observations both from attempting to teach other software analysis techniques and having design debates with experienced professionals. All too often design debates start from different designs all of each exhibit good principles, practices, and patterns then move into relative merits of specific technologies without ever reaching a rigorous comparative analysis. Similarly our students frequently have trouble applying simple, but somewhat subjective, judgments about 'uses', encapsulation vs. separation, and other principles. The lack of precise, objective building blocks for analysis results in considerable confusion before we even get to the cognitively difficult tasks of analyzing impacts and making tradeoffs.

The technique we developed 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. Dependencies into two categories: '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.

In this paper we present our results teaching this technique to a group of 30 undergraduate information systems students. In one and a half hours we were able to explain the technique, discuss the implication and uses, and give some examples. We gave a thirty minute test covering 13 questions followed by a usability survey based on the Likert-style survey developed by Brooke[10]. The test showed that the majority of the students were able to learn and correctly apply the technique to a variety of questions and cases. Where students had difficulty we saw the same mistake repeatedly, suggesting that slight improvements to the teaching could result in nearly perfect application of the technique. The usability survey indicated that students generally found the technique easy to use and would use it in the future; on the negative side they felt it required learning a lot in a short time frame and were not extremely confident in its use.

This study is similar to the work on DSMs in education [11, 12] in that 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 also checks students' ability to relate implementation to design level dependencies, but this study focuses primarily on design questions – where will ripple effects occur, which alternative design will best handle a given change, and how can expected variations be protected.

The rest of this paper is organized as follows: Section 2 explains the dependency graph technique. Section 3 discusses the method of the study and questions used for evaluation. Section 4 discusses the results and Section 5 concludes.

## 2. Dependency Graph for Modifiability Analysis

This new technique extends on Parnas's early concept of 'uses' structures, Jackson's analysis of assumptions, indirect coupling and design structure matrices. 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 views systems as being composed 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. In an object-oriented language a class is the most common example, but could also include a configuration file when it specifies system behavior, or a package of classes. Data is defined as any variable, or information storage not including behavior or processing. An interface is defined as a precise description of how to access the behavior of a module while being separate from the implementation of that module – for example a java interface. The behavior nodes are useful in comparing designs with different modularizations, but in practice it is frequently possible to omit them and include only modules and interfaces.

The model has two kinds of dependencies to show relations between the element. Behaviors can depend on other behaviors or on data through 'semantic' dependencies. These represent the need for functions to get data with the proper semantics – similar to Yang's indirect dependency[8] – and also capture the intent of Jackson's assumptions[7] by one function about other behaviors. We call these semantic dependencies because of the dependence on the right meaning of the data as separated from the details of its format or method of access.

The second category of dependencies is syntactic dependencies. These arise from the syntax of the implementation code. We break these into three types. First, a module can implement an interface. This indicates a promise by the module that any client which uses the rules of the interface to access the module will get the results promised by the interface. This implies that a change to the definition of the interface, without a corresponding change to the module, results in a module which no longer fulfills this promise. The implementation of the interface – the module – can be changed, even replaced, without breaking the promise as long as the new implementation continues to provide the results specified by the interface.

A module may also depend on an interface by having a reference to that interface. This represents a direct access to a behavior defined in the interface and can be observed as a code reference within the module to that interface. If the interface is modified, the referencing module will also need to be modified in order to work as before.

Modules may also depend on other modules through references. Like references to interfaces, this represents direct access from one module to another. This can be observed in code as reference or other explicit naming of the depended on module. A module to module reference implies that a modification to the depended on module may require a modification to the dependent module.

The last dependency type between modules is a data dependency. The module access data whose type and format is defined by the module depended on. This dependency is different from Yang's indirect dependency, in [5] the "definition" of the data is wherever the data is first instantiated while here the "definition" is where the format expected by the receiving module is defined. A data dependency may "pass through" several modules (for example if an xml document is created and then passed through several intermediaries before being parsed), but often is only direct (module X gets an array of integers from module Y – how module Y gets those integers is unknown to X). The dependency between the actual source and sink is captured by the semantic dependency described earlier.

The full set of nodes and dependencies are represented graphically as shown in Figure 1.

Given this graph structure we can assign modifiability properties to pairs of nodes based on the dependency or lack of dependency between them. 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. Making behaviors changeable with respect to each other requires structuring the behaviors themselves such that they are either not dependent on each other's data or ensuring that the data's semantics are not changed when the behaviors change.

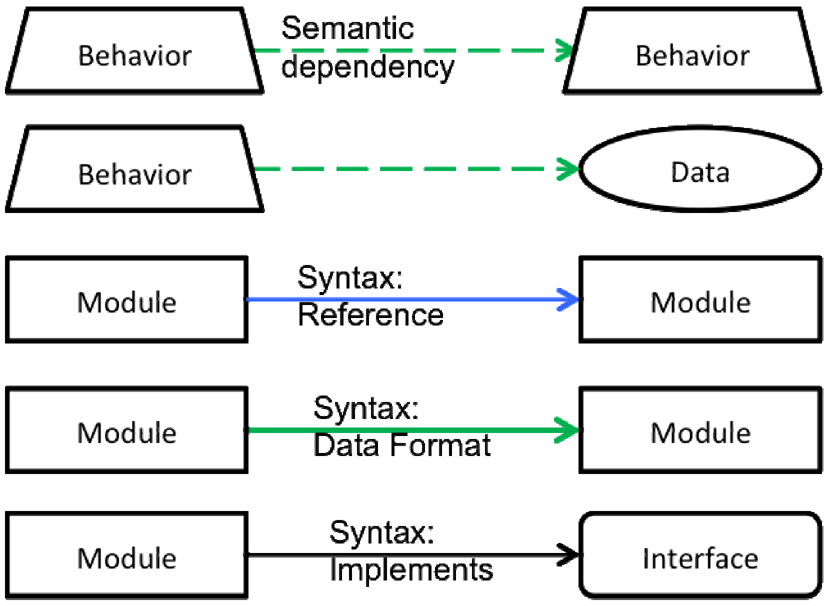

**Figure 1: Nodes and dependency relations**

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.

Assuming no semantic changes to data, we can then assign properties to modules based on the syntactic dependencies. The "Changeable" property derives from the idea of protected variation – that other modules are protected from variation in the "changeable" module. Thus changeable" is directional and relative – module A is "changeable" relative to B does not imply B is "changeable" relative to A nor that A is "changeable" relative to C. *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 results in modifying larger and more complex modules. Note that part 2 implies that syntactic dependencies are considered non-transitive. The intuition is that if Z depends on Y and Y depends on X, then modifications to X which effect the dependencies from Y to X can by absorbed by Y and need not impact the syntax Z uses to access Y.

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 the protected set. Thus given a modification to the seed module, all modules in the dependent set may need to also be modified, while modules in the protected set will not have to be modified.

In figure 2 we show the dependency graph for a potential design of a vending machine. In this design the Coin Collector module identifies inserted coins and calls the Coin Return to increment its count and holdings. The Coin Collector also activates the panel with the amount of money inserted. The Selection Panel highlights the sodas available and calls the Soda Dispenser with the selected soda. The Soda Dispenser emits cans and informs Soda to change the inventory. Based on the price from Soda the Dispenser tells the Coin Return how many coins to return. Focusing on the bottom half of the graph, we can see that the Soda Dispenser makes calls to both Coin Return and Soda and hence has syntactic dependencies to both. Soda Dispenser is called by the Selection Panel and thus is depended on. Selection Panel doesn't do anything with the output of the dispenser so there is no semantic dependency there. However, the 'return coins' behavior of Coin Return depends on the Soda prices, no matter how those prices get to Coin Return, hence the semantic dependency.

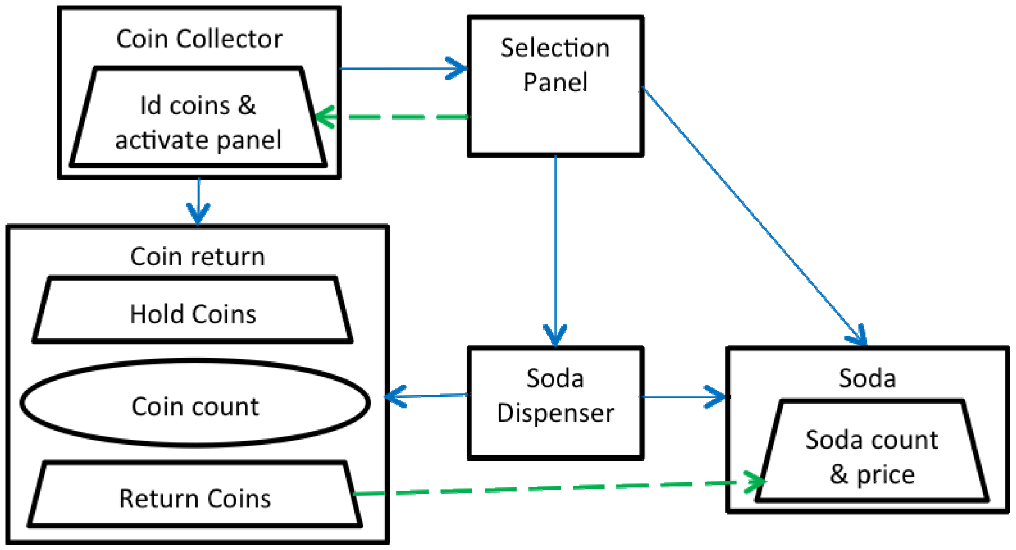

**Figure 2: Dependency graph for a potential design of a vending machine**

We are going to test whether students find this technique easy to learn and apply. As the syntactic dependencies have a direct mapping to code, we expect that students with some development experience should have no trouble understanding the concept of dependencies. Additionally, since the modifiability properties are deterministic, we expect that even without extensive experience students will apply the technique correctly. Our study will assess how well students are able to learn and apply the technique after a short training workshop.

## 3. Study Methodology

For this study we got thirty student volunteers to sit for a 1.5 hour workshop on the technique. All were in either their third or fourth year of an undergraduate information systems degree. All had completed courses on programming, software engineering, databases, and systems integration, including multiple large (semester long) team projects. Seven of the students had completed a group, capstone project of at least four months. Thirteen had completed an internship with significant programming work of at least 10 weeks. Based on our previous classrooms experiences with the students, we estimate they were distributed across the top half of the cohort. We obtained approval from our Institutional Revenue Board for this human subject study before running the study.

The workshop consisted of a lecture portion including motivation for the technique and covering the dependency graph technique 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. Finally we discussed example cases using the technique to decide between alternative designs and using the technique to improve designs.

Immediately after the workshop we had the students attempt a 13 question 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 after limited training. We had five categories of questions to assess five tasks the students should be able to complete. First was to take sample code and produce a dependency graph showing the syntactic dependencies in the code. The second was to take a given sequence diagram and produce a dependency graph to match. 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. 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 with fewer ripples. Lastly, we presented them with a candidate design and change scenario and had them modify the design to make the change easier to accommodate. The first two categories let us measure how well students are able to match implementation and design with dependencies. The third let's us measure the students' grasp of the graph properties. The last two categories test whether students can evaluate the implication of those properties on decision making.

After the assessment students were asked to fill out a ten question usability survey. This survey was a reworded version of Brooke's System Usability Survey[10]. The survey uses a Lickert scale to elicit responses on difficulty in learning a system, need for expert assistance, likelihood of future use, complexity and confidence in use. We used this survey to assess the students' perception of the ease of learning to apply this technique.

## 4. Results

Across all 13 questions the median student answered 10 correctly. The distribution is shown in Figure 3. In Table 1 we show the percentage of correct answers by category of question. We discuss these results in more detail below.

**Table 1: Percentage of correct answers by question category.**

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

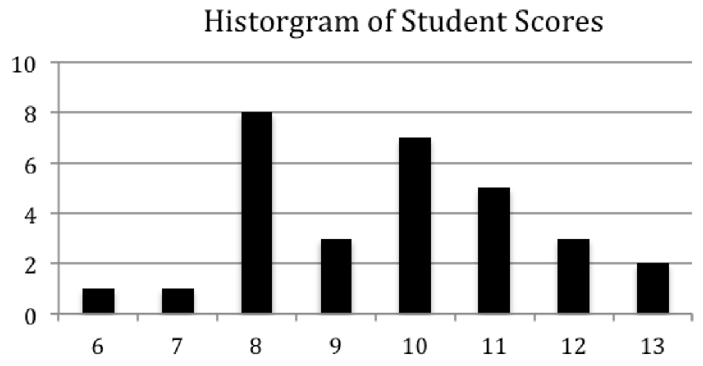

**Figure 3: Number of students vs. correct answers**

Interestingly, the best results were for the questions on choosing between alternative designs, which was our main goal for the technique. Across the three decision questions, on average 88% of the students were able to choose the design with fewer ripple effects for the proposed change. If we omit the two students who did not complete all questions, the percentage rises to over 91%. We take this as indication that the technique is clear and straightforward for making comparative analyses. The two questions on improving existing designs showed 85% were able to apply strategies from the workshop to decouple problem dependencies.

The students also did extremely well in translating UML diagrams to dependency graphs, with 92% of the answers correct. This is highly encouraging as it suggests that students could effectively work with the technique while thinking through their designs – the appropriate time to consider modifiability implications.

The questions requiring translation from code to dependency graphs caused more trouble with only 62% answered correctly. The lower accuracy could be due to either a misunderstanding of the precise relationship between code and the dependencies represented in the graph, or due to a lack of careful attention in reading the provided code. The latter explanation is not much concern; since the mapping of code to dependency graphs is completely deterministic, this step can be automated – and we have done so for one large (~1 million LOC) system. If we were to use this technique more extensively in programming courses we would likely give students an automated tool. Of more concern is the possibility that students do not see the linkage between code structures and dependencies inhibiting modifiability. This is the same question explored in [11]. The disconnect between high level understanding of dependencies and low level code syntax instantiating those dependencies is an area of concern we hope to investigate further in the future.

The most difficulty was encountered in using provided graphs to identify likely ripple effects for a given modification, with 53% of questions answered correctly. However, those making mistakes nearly all made the same error. Based on the repeated mistakes we could make slight changes to the workshop and expect over 70% accuracy.

One common error was misinterpreting the arrow directionality. We have chosen for arrows to follow the direction of dependence. Thus A→ B means A depends on B. However this means that ripple effects go the opposite direction of the arrows – modifications to B could ripple to A. We've found that in analyzing the graphs some students interpret the arrow direction inconsistently and thus expect different effects than the graph indicates. While we were quite explicit in the workshop about the meaning of the arrow direction, we could do more exercises to solidify the understanding. A second common mistake comes up when the scenario calls for adding a new module. If we are not explicit about how the new module is placed in the design, students often make surprising assumptions that do not parallel the original design. This issue impacted one question in the category for identifying ripples.

For the survey we got an average score of 70, which indicates a generally favorable view from the students. The highest score was for “I would like to use this technique frequently” – mean 4.2 out of 5 – which suggests that they found this technique helpful in comparison to the informal and subjective methods they learned in previous classes. The lowest score – average 2.9 – was for “I needed to learn a lot of things before I could get going with this technique”, indicating that the concepts were new or challenging to roughly half the students.

Notably, we did not have students complete any exercises and get instructor feedback during the workshop period. As a result, the post-workshop test was done ‘cold’ as a very first attempt at using the technique. In retrospect this was a mistake. We know that even short exercises plus feedback eliminate many simple misunderstandings and leads to much better learning. As we’ll discuss under the results a couple of simple misunderstandings appeared in the post-workshop test which could have been avoided. We use exercises extensively in our normal courses and will add them any future workshops on the dependency graph technique.

We have two concerns about the results from the test. Many of the students had just completed a class with the author. Their responses to the usability survey may have been influenced by a desire to please their instructor. Lastly students had more difficulty as the dependency graphs got more complicated. Our experience with industry systems suggests that simplicity can be maintained, but that graphs can get very large, and without care portions become complicated. Thus with complex systems more automation and tool is likely to be necessary to make sense of the graphs.

## 5. Conclusions

The paper reported the teaching of a new dependency graph technique for analyzing software modifiability to third and fourth year undergraduate students. 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 only a short workshop. Additionally the students generally felt the technique was easy to use and would be likely to use it in the future. This study demonstrates that objective design analysis can be taught to students quickly.

### Acknowledgment

The author would like to thank Singapore Management University for supporting this work. The author would also like to thank Eric Nyberg from Carnegie Mellon University, Narayan Rammasubbu from University of Pittsburgh, and Jason Woodard from Singapore Management University for extensive advice in formulating both the technique and the study.

## References

[1] Parnas, D.L., “On the Criteria To Be Used in Decomposing Systems into Modules.” Communications of the ACM, 1972. 5(12).

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

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

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

[5] Gamma, E., et al., “Design Patterns: Elements of Reusable Object-Oriented Software.” 2002: Addison-Wesley.

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

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

[8] Yang, H., E. Tempero, and R. Berrigan, “Detecting Indirect Coupling”, in 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] System Usability Scale from http://www.usabilitynet.org/trump/methods/satisfaction.htm

[11] Y. Cai, R. Kazman, C. Jaspan, J. Aldrich. “Introducing Tool-Supported Architecture Review into Software Design Education.” In Proc. of the 26th IEEE Conference on Software Engineering Education and Training. 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.