Introducing Tool-Supported Architecture Review into Software Design Education

Yuanfang Cai[α], Rick Kazman[β], Ciera Jaspan[γ], Jonathan Aldrich[λ],
Drexel University[α], University of Hawaii[β], Cal Poly Pomona[γ],
Carnegie Mellon University[λ]
yfcai@cs.drexel.edu[α], kazman@hawaii.edu[β], ciera@csupomona.edu[γ],
aldrich@cs.cmu.edu[λ]

Abstract

While modularity is highly regarded as an important quality of software, it poses an educational dilemma: the true value of modularity is realized only as software evolves, but student homework, assignments and labs, once completed, seldom evolve. In addition, students seldom receive feedback regarding the modularity and evolvability of their designs. Prior work has shown that it is extremely easy for students and junior developers to introduce extra dependencies in their programs. In this paper, we report on a first experiment applying a tool-supported architecture review process in a software design class. To scientifically address this education problem, our first objective is to advance our understanding of ***why*** *students make these modularity mistakes, and* ***how*** *the mistakes can be corrected. We propose tool-guided architecture review so that modularity problems in students' implementation can be revealed and their consequences can be assessed against possible change scenarios. Our pilot study shows that even students who understand the importance of modularity and have excellent programming skills may introduce additional harmful dependencies in their implementations. Furthermore, it is hard for them to detect the existence of these dependencies on their own. Our pilot study also showed that students need more formal training in architectural review to effectively detect and analyze these problems.*

1. Introduction

Modularity is arguably the single most important design principle in software engineering. Many of the advances in software design and architecture over the past four decades have been modularity-related: abstract data types, object orientation, aspects, and services, to name but a few. The primary purpose of these constructs is to create software that is *easily evolvable* and adaptable to changing requirements.

The teaching of software modularity thus poses an interesting dilemma: the true value of modularity is often realized only as software evolves, but student homework, assignments and labs, once completed, seldom evolve. Therefore, students never receive any indirect feedback regarding the modularity of their designs: they do not see their design decisions realized as easier-to-maintain—or harder-to-maintain—software. Students also do not receive direct feedback: unlike other software properties, such as functionality and performance, there is no rigorous, widespread way to assess modularity and evolvability. Therefore, important design principles such as information hiding and separation of concerns are often conveyed to students informally and students are seldom assessed on them. Worse still, students' understanding of these principles is not grounded in the actual experience of evolving software. Even if an instructor wanted to evaluate whether students were following modularity principles and if their code could cause future maintenance problems, it is difficult to do so, in part due to the lack of good methods to detect poor modularity decisions. As a result, new software engineers in industry do not pay enough attention to, or do not know how to achieve, software modularity, leading to high maintenance costs in practice.

978-1-4673-5140-9/13 © 2013 IEEE CSEE&T 2013, San Francisco, CA, USA

Recent research [3][11] has revealed that it is extremely common for programmers to produce functioning, but poor quality code with numerous unexpected dependencies, *code smells* [7] and *technical debt* [5]. Cai et al. [3] describe an experimental study in which students were given a series of design tasks, implementing UML-specified designs that make heavy use of design patterns [8]. The authors noted that as many as 74% of student submissions did *not* faithfully implement the provided modular design, even though their software functioned correctly. Although one of the purposes of applying design patterns is to improve software modularity, students in our study introduced extra dependencies (an example of technical debt), unnecessarily coupling modules that are supposed to be independent. This unnecessary coupling was introduced even though we gave the students small problems (ones that can be solved in an hour or two of work) and a pattern-based solution, described in industry-standard UML notation.

These results, together with the widespread existence of code smells [7], technical debt, and modularity decay in industry, are troubling for the practice of software design. We believe that these problems, which are widely observed in real world, have their root in deficiencies of software design education.

Additionally, part of the problem appears to be due to disposition, not due to lack of knowledge: a separate study at CMU found that students thought it was important for their code to depend on the implementation details of other code [16] which is a major modularity concern. We hypothesize that students may not recognize these issues as design problems because such dependencies typically do not have immediate consequences. It is not until the software is being maintained and evolved—months or years later—that the (often dire) consequences are manifested. This is when the technical debt must be repaid.

To scientifically address this education problem, the first objective of our work is to advance our understanding of *why* students make these modularity mistakes, and *how* the mistakes can be corrected. To do this we have introduced a tool-supported architecture review [4] approach into undergraduate software design education, based on our previous research. We hypothesize that the architecture review process will guide students to evaluate their design against possible future change scenarios, and the tool, based on a *design structure matrix* [1] representation, will facilitate this review process by making the modular structure of a design and its associated code explicit.

To introduce this approach into software design education we furthermore had to create pedagogical design problems, model solutions for these problems, and associated architecture review questions and solutions. In addition, to understand effectiveness of our approach, we also designed surveys that attempt to gauge the students' (possibly changing) attitudes towards modularity. Using these teaching materials and surveys, we are conducting a series of experiments in each of the authors' universities. We expect that the materials, surveys, and processes will be refined after each experiment. In this paper, we report on the first experiment applying this process in a software design class at Drexel University. As a pilot study, this experiment was also meant to assess the feasibility and effectiveness of the new materials, assessment instruments, and process.

Although the number of students who participated in the experiment was not statistically significant, the results consistently pointed in a clear direction. The major findings of this research include:

(1) Even students who understand the importance of modularity and have excellent programming skills may introduce additional harmful dependencies in their implementations.

(2) It is hard for students to detect the existence of these dependencies without an effective review process. That is, when students were only given the tool and model design, they could

tell that their design was different from the model solution, but could not understand *why* the extra dependencies were harmful.

(3) Students need more formal training in architectural review, to effectively detect and analyze these additional dependencies.

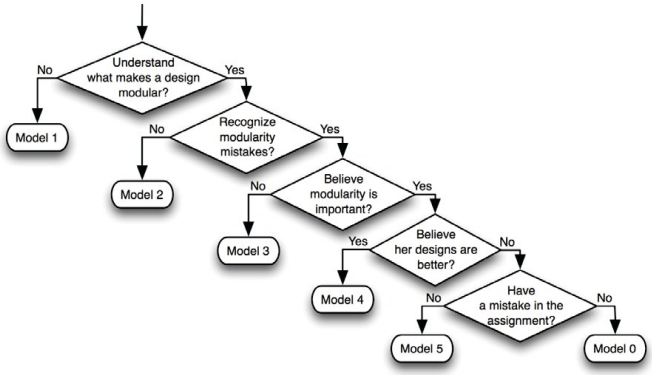

Figure 1: Decision Diagram for Student Mental Models

2. Understanding the Root Causes of Mistakes

First of all, we need to understand why students introduce extra dependencies. Is it because they don't understand the concept of modularity, don't think modularity is important, or simply have difficulty following design principles? Based on our prior experience [3], we hypothesized five possible reasons for the extra dependencies, and designed a pre-survey and a post-survey that, when combined with architecture review results and the comparison between student submissions before and after the review, allow us to follow the decision diagram in Figure 1 to validate these hypothesized reasons and assess each student's mental model.

1. Poor knowledge and understanding of modularity. The student does not understand the relevant design principles and does not realize that she introduced extra dependencies that defeat the modular structure. To assess this model, we used a short set of exam-style survey questions focused on knowledge components of modularity at the two lowest levels of Bloom's taxonomy [17]. Students who fail to understand these concepts are likely to produce designs with mistakes simply because they do not understand the modularity principles that should drive their design. Students who understand modularity are either not making design mistakes, or are making them for other reasons. Below are two sample survey questions:

1. Which statement do you believe is more true?
   a. It is important to depend on the internal details of another class.
   b. It is important for APIs to contain only as much as the user needs to know.
2. Which statement do you believe is more true?
   a. It is important to divide up code based upon function in order to make it maintainable.
   b. It is important to make details public so that others can change them when necessary.

2. Poor skill in recognizing modularity-related implementation mistakes. The student understands design principles, yet he cannot effectively recognize mistakes made when a design is incorrectly realized in code. Ideally, this would be assessed by showing students a design, along with code examples that correctly and incorrect implement that design, and determining whether they can find the problems. In the first experiment, we took a simpler approach, comparing students' implementations with our model design, and making a judgment based on their programming skills, assuming that students with good programming background should have little difficulty realizing a given design. We also used survey questions asking about how much effort they spent to make sure their code follows the design, and whether they thought their code confirms to the design.

3. Indifferent attitude towards modularity. The student understands modularity and can apply it, but she believes that it doesn't matter. She might think that the only thing that matters is to make the software work and pass all the functional tests. In this pilot study, we used a survey question to ask students if they think modularity is important. In the future, we will use a

set of forced choice questions similar to [16] to gauge whether the student believes that modularity is an important quality attribute in code. If the student does not believe in modularity as a goal, this may explain any mismatches against instructor-furnished designs. Students who do believe modularity is important would have produced mismatched designs for other reasons.

*4. Belief that a self-created design is better***.** The student fully understands modularity and believes it is important, but he thinks that his design is better than the instructor's design. For example, he might think that the negative consequences of the changes he is making are insignificant, or that he has made a better tradeoff choice in his design. In this pilot study, we used a survey question to ask the students whether they think their design is better. In the future, we will have a series of Likert-style survey questions regarding properties of the instructor's design, properties of the student's design, and a comparison of the two.

5. Unintentional mistakes. The student fully understands modularity and believes it to be important, yet makes mistakes for other reasons. In the pre-survey, we asked the students whether they think their code conforms to the given design. In the post-survey, we asked if they found any extra dependencies in their implementation. If the answer is yes, then we further asked if these extra dependencies were surprises. For students who have reached this point, it is possible that any mismatch between their designs and the instructor's design was a simple mistake due to oversight or time constraints, and not from a fundamental misunderstanding of the material.

The outcome of any of these models will be a mismatch between the students' produced designs and the expected (ideal) design. It is important for us to distinguish between students with these different mental models so that we can adjust our instruction accordingly.

3. Pedagogical Materials for Architecture Review

In our previous study [3], we observed that students' mistakes are not primarily concerned with the large-scale structure of the patterns, but rather with the implementation details that contribute to each pattern's modularity effects. Our observations suggest that students learn the gross structure of patterns easily, yet they make mistakes in applying this structure while writing code or designing a new system. Many students followed the high-level structure of design patterns, but made lower-level mistakes that violated the patterns' intent.

We propose a comprehensive approach to building mental connections between design patterns and modularity principles through the use of *architectural reviews ([2],[9])*. An architectural review is a structured analysis of a design, focusing on how it fulfills quality attribute requirements. In our case, we are interested in the quality of modularity, and we are looking at the pattern level of design (micro-architecture). The structure reverse-engineered from the code reflects the student's design decisions. The goal of these reviews is to force students to think not just about design structure, but design *intent*, considering possible future changes. To this end, we created (1) pedagogical design problems and model solutions, (2) evolution scenarios associated with these problems, (3) *design structure matrices* (DSM) that reveal the modular structures. Next we explain these concepts and demonstrate how DSMs can be used to facilitate architecture review.

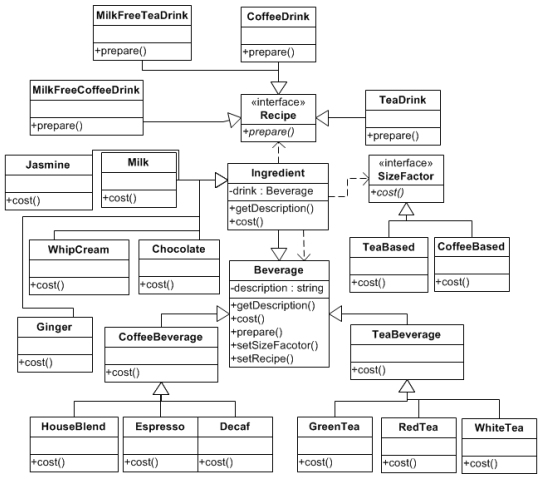

Figure 2: Extended StarBuzz Coffee System

1. Pedagogical design problem. In the pilot study, we used a lab assignment modified from the StarBuzz system [6]. The purpose of this lab is to apply a combination of decorator and strategy patterns. Figure 2 depicts the UML class diagram of this design. The decorator pattern is used to calculate the cost of a drink with an arbitrary number of ingredients, dynamically added in any order. The first strategy pattern is used to add the cost associated with size, and the second strategy pattern is used to print the recipe for different types of drinks: coffee, tea, milk-free tea, or milk-free coffee. The system automatically calculates the cost of a drink and prints out a recipe for it. In the lab, the students are given the functional requirements, as well as a UML class diagram that shows the class structure of the system and the most important methods and attributes, but does not include details such as complete method signatures.

2. Evolution Scenarios. The purpose of applying design patterns is to facilitate future evolution of the system. The modularity and evolvability of a design should thus be assessed against possible future change scenarios. For the StarBuzz system, we considered 5 change scenarios. We ensured that the model solution, developed in Java, faithfully implements the designed modular structure so that under each change scenario, the changes to the code are either localized or follow the open-closed principle [13]. That is, a minimal number of existing classes need to be modified, or classes are added without changing existing classes.

The five change scenarios were designed to test the students' ability to implement the intended modular structure, rather than simple change requests, such as adding a new ingredient. Figure 3 shows such an example. Our purpose was to gauge the students' ability to not introduce extra dependencies, which would defeat the original modular structure.

Scenario 1: The new requirement is that if a drink does not have any ingredient, then the program doesn't need to print out any recipe.
a. How to revise your design/code to accommodate this change:
Which and how many of classes to be added, deleted or changed?
b. Do you think your design/code can be improved to better accommodate this change? Yes No
If the answer is yes, please explain:

Figure 3: Sample Scenario in the Architecture Review Form

We created an architecture review form, following the format above, to guide the students in evaluating their design and the change impact of the five scenarios. The students were asked to submit the completed form after the review process.

3. Model Design Structure Matrix. To facilitate the review process, we leverage a DSM to visualize the modular structure and quantify the impact of a change. Figure 4 illustrates a DSM reverse-engineered from our model Java program. A DSM is a square matrix; its columns and rows are labeled with the same set of elements in the same order. If an element at row *x* depends on an element in column *y*, then the cell (*rx*, *cy*) will be marked. The DSM in Figure 4 depicts 25 classes grouped into multiple modules—the inner blocks along the diagonal with grey backgrounds and dark borders. This DSM is clustered to manifest the following modular structure:

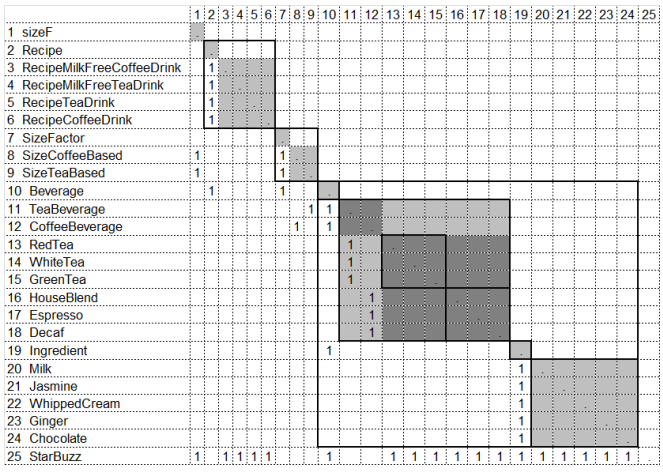

Figure 4: StarBuzz Model DSM

The first element, *sizeF,* is an enumeration type for different sizes. The second module (r2-6, r2-6) shows the recipe module, led by the *Recipe* interface in a strategy pattern. A client of this pattern should only interact with this interface (a key *design rule*, in Baldwin and Clark's definition). Similarly, the third module (r7-9, c7-9) is another strategy pattern module. The *Beverage* class (r10, c10) is a key design rule of the decorator pattern, leading 8 classes of tea or coffee beverages. Another key design rule of the decorator pattern is the *Ingredient* class (r19, c19), which is a subclass of the *Beverage* class (thus the dependency in (r19, c10)), and leads the module containing 5 ingredients (r20-24, c20-24). The last one is the control class *StarBuzz* that reads user inputs. If a student implements the design faithfully, when her code is reverse engineered into a DSM, we should see the same, or similar, modular structure.

We use DSMs in our experiment to facilitate architecture review and assess the change impact quantitatively. For example, in the first scenario as shown in Figure 3, the new requirement states that recipes should not be printed for beverages without ingredients. From the DSM, we can see that only the *Beverage* class and the *StarBuzz* class depend on the *Recipe* interface. As a result, it is easy to tell that changes can be limited to these two classes in order to add such a constraint. Given a DSM reverse-engineered from a student's submission, we detect extra dependencies by comparing how it differs from the model DSM. Each student was provided with a DSM generated from their code so that they can quantify change impact in their architecture review process.

4. The Experiment Procedure

Our pilot experiment was executed in a junior level software design class at Drexel University. The students include CS and SE majors with different levels of programming experience. To ensure that all students had a minimal level of Java competence, we conducted the experiment in the 8th week when all important modularity concepts had been introduced, and the students had done a series of Java programming labs and homework assignments. The experiment was conducted after the students submitted their Starbuzz lab solutions.

Before they submitted their Starbuzz lab assignment, we first asked the students to complete a pre-survey to assess their disposition towards modularity, the effort spent to make the code modular, etc. The instructor (the first author) then spent about 30 minutes in class introducing the DSM concept, the architecture review process, and how to leverage DSMs in this process. After that, the students were divided into 5 groups as explained below to conduct the architecture review. After the review, students were asked to turn in their review form, to resubmit their lab if they made any changes to fix the problems found during the review, and finally to complete a post survey.

Architectural reviews can be deployed in multiple ways, and one of our goals is to better understand the most effective way to use them in teaching. It is our hypothesis that using some form of architecture review will help students understand modularity, but it is not yet clear how to best structure these reviews. Therefore, we created four experiment groups (plus the control group) to investigate these options: self-review of student programs with a model DSM, self-review without a model DSM, instructor-review of student programs, and peer-review of student programs. Each of these groups is described below.

Control group (3 students). Students in the control group completed the assignment without any formal design review. They were given the lab instructions and a UML class diagram, and they were graded based upon how well their implementation matched these artifacts. The control students had the same amount of time to complete the assignment, and they received the same instructions as the other groups, including the grading rubric.

Self-assessment with model DSM (4 students). Students in the self-assessment group with a model DSM were given a review form and both a model DSM and a DSM reverse-engineered from their code (which we call a *sample DSM*). The differences between their implementation and the model implementation are marked in their DSM. As illustrated in Figure 5 showing a sample DSM, the cells with dark background and white font indicate extra dependencies introduced by the student that do not exist in the model DSM. The black cells indicate dependencies that are in model DSM, but not in the student's sample DSM.

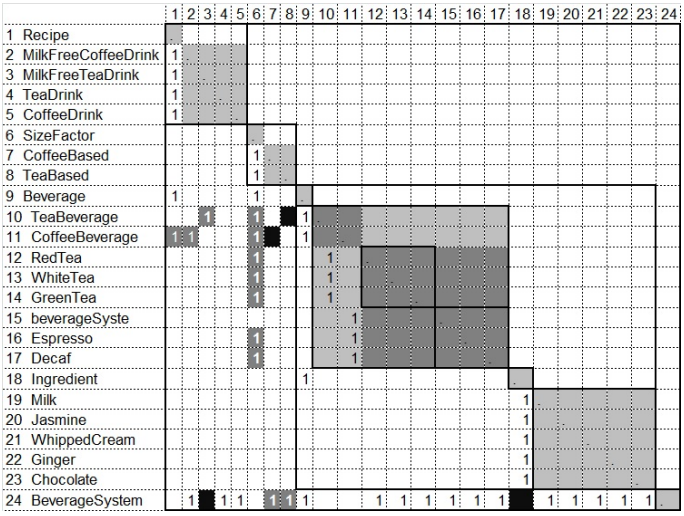

Figure 5: A Student Implementation DSM

Self-assessment without model DSM (3 students). Students in this group also received a review form and a sample DSM generated from their code, but not the model DSM, nor the differences between them. This reflects a more realistic situation where developers can use reverse engineering tools such as Lattix[1] to generate a DSM, but are not given an ideal solution.

Instructor-guided review group (3 students). Students in the instructor-guided group met with the instructor who conducted the design review. In this condition, the students did not use the DSM tool themselves; the instructor used the tool to evaluate each student's assignment in advance. During the meeting, the students were asked to read the review form, discuss with the instructor whether their current code can accommodate these changes locally or incrementally, following the open-closed principle. They looked at the sample DSM together to analyze how these changes will impact their code.

Peer-assessment (2 students). Students in this group paired up and used the DSM to evaluate each other's assignment, without the model DSM. They were supposed to follow a process similar to the instructor-guided review and complete the review form together.

All groups, including the control group, were given the opportunity to fix their lab assignment and resubmit in week 10. Students in experiment groups were also asked to turn in the review form. After the resubmission deadline, we conducted a post-survey to see (1) if the students found any harmful extra dependencies, if so (2) whether these extra dependencies were surprises to them, and (3) whether they fixed these extra dependencies in their final submission. We estimated that the review process should take about 30 minutes, but the instructor-guided review took from 45 minutes to 1 hour. As a lesson, next time we should ask the students to record the time spent on the review.

5. The Results

Since this was our first test of our experimental protocol, we were more concerned with qualitative than quantitative results. As such our numbers of participants were not large: there were 23 students originally registered the class, 4 dropped due to lack of Java background. Of the remaining 19 students, 15 agreed to participate in the experiment. Of these 15 students, 14 completed the pre-survey, all participated in the architecture review, 8 of them submitted the architecture review form, and 10 completed the post survey. Note that 3 of the 15 students participated in the experiment but didn't submit the form because they were in the control

[1] http://lattix.com/

group. Four students in the self-review groups (three without model and one with model) claimed that they are too busy to finish the experiments. We understand that, given the small number of students, we cannot claim that the following results are statistically significant. However, the results are consistent and conveyed interesting factors that need to be considered in next rounds of experiments. Next we report the results on student dispositions and DSM-based architecture review effectiveness:

5.1 Student Skill and Belief on Modularity

We infer the mental model of students by combining the results of the pre- and post-surveys, the architecture review form, and the student submissions before and after the architecture review experiment. We also combine these observations with the student's class performance.

Model 1: Knowledge and Understanding of Modularity. For the 8 questions designed to test the students' understanding of modularity in the post survey, we observed that the 3 students who received the worst score in the class answered 3 of the 8 questions wrong. In addition, even good students believed that "APIs should contain a lot of details". Retrospectively, we suspect that there were ambiguities in the wording of the questions, which we will clarify in our next experiment. Considering the fact that the experiment was conducted in the 8th week, students with reasonably good grades should have decent understanding of modularity.

Model 2: Skills in applying modularity. Again, the level of skill is associated with the student's overall performance in the class. According to the pre-survey, the 3 best students in the class, who we are sure have the skills to realize the design, claimed that they spent a lot of effort to ensure that their code was consistent with the design. 7 students claimed that they are aware of some discrepancies, but largely followed the design. 2 students claimed that their code doesn't follow the design. These 2 students happen to be the ones with worst grades.

Model 3: Disposition about the Importance of Modularity. According to the pre-survey, all but one student thought modularity is important: 7 students scored it 4 (very important), and 6 scored it 5 (extremely important). The next section discusses threats to validity of these results.

Model 4: Belief that Self-created Design is Better. Only one student thought his design is better than the given design. This was not necessarily unjustified, because his sample DSM showed that his code had exactly the same structure as the model code.

Model 5: Unintentional Mistakes. In the pre-survey we asked: if they think their code is different from the given design, what could be the reason. 7 students answered this question. 5 of them claimed that they were in a rush and didn't have time to check the conformance. 1 student claimed that his code is better. 1 student claimed that he doesn't think it matters that the code conforms to the design. Interestingly, the DSMs of these 2 students are exactly the same as the model DSM. Moreover, of all 11 students who thought that they made sure their code followed the design (according to their answers in the third pre-survey question), only 2 of them indeed followed the designed structure; 2 of the 3 students who thought their code strictly followed the design actually introduced many extra dependencies.

The experiment showed that, for students with weak programming backgrounds, who performed poorly in class, it is hard for them to simply follow a given design in UML, even though they think modularity is important (all three of them claimed that modularity is extremely important). For the majority of students with sufficient background and skill, although they think modularity is important and spent a lot effort on it, they still introduced extra dependencies unintentionally.

5.2 Architecture Review Effectiveness

After reading the submitted architecture review forms, and comparing their submissions after the review, we made the following observations:

1. The instructor review group had the most striking results: the students did not realize that they had introduced extra (harmful) dependencies until we did the review, at which point they were able to correctly repair their code. For example, the DSM in Figure 5 is a sample DSM for a student from this group. This is one of the best students in the class, who has solid programming skills. In his pre-survey, he indicated that (1) he believed that modularity is very important, (2) he spent a lot of effort making the sure the code followed the design, and (3) he believed that his code completely followed the design. However, the DSM shows that his code introduced more than 10 extra dependencies. By considering Scenario 1 as shown in Figure 3, he realized that, to accommodate this change, he had to change at least two additional classes: *CoffeeBeverage*, and *TeaBeverage*, due to the extra dependencies shown in cells (r10, c3), (r11, c1) and (r11, c2), which were unintentionally introduced. He figured out that these dependencies were extra when the instructor pointed out that this change cannot be accommodated locally, and asked if the code can be improved to reduce the impact.

Interestingly, another member of this group had already reverse-engineered his code into a DSM using Lattix, but he didn't realize which dependencies were extra or harmful until we conducted the review. The implication is that architecture review can be an effective way of reducing unintentional structure mistakes, if (1) one is proficient in architecture review and (2) one understands the intent of the design.

2. The peer review group found extra dependencies, but they believed that using some other patterns learned in class would be the solution, so they did not subsequently fix their code. The implication is that only having a design analysis tool (such as the DSM) does not mean that the student can fix problems effectively without additional support and education.

3. One self-review-with-model group member found problems after the review, but her code had too many problems and she did not fix all of them. This implies that the experiment only makes sense for students with reasonable programming background. The other 3 members didn't submit the form or fix the extra dependency either because of time pressure, or because they were not sure how to conduct the review given just a 30 minute lecture. Even though they saw differences between their code and the model, they were not sure why these extra dependences were harmful. This, again, implies that just having tools is insufficient, and substantial guidance on architecture review is needed.

4. Of the three self-review-without-model group members, one didn't submit the review form. The other two were neither able to tell whether their code followed the design, nor to identify extra dependencies by just looking the DSM. For example, if the student had a sample DSM is like the one in Figure 5, but without the contrast, he might see that, e.g, there are 10 dependencies in column 6, but would be unable to determine if this is a problem.

5. The control group did not fix *any* modularity issues in their resubmissions, implying that the architecture review appears to be making a substantial difference.

In summary, it appears that most students cannot recognize modularity mistakes without external input and assistance. They were largely unable to recognize their modularity mistakes simply by being given the model design (in UML) and model DSMs. The results implied that most students, while they understand modularity principles in theory and have sufficient skills, need considerable assistance to understand how these principles apply to their design and code, and to appreciate the implications of violating modularity.

5.3 Threats to Validity

The first threat to validity of this study is the small number of students. We will repeat this experiment at Cal Poly Pomona and CMU in the near future to see if the results vary. This pilot study may not be sufficient to test students' mental models because the students may just chose "modularity is important" to please the instructor. The lack of efficiency of the other groups

may also due to the fact that the lecture about architecture review was too short and the students needed more time and exercise to get better with it. Other threats to validity include the wording of the surveys or the design problem itself. We will explore other options to see if the results will differ. We may also need to revise the mental model because we observed that some students have more than one model. For example, a student without sufficient skill may still think modularity is important.

6. Conclusion and Future Work

We have reported a pilot study introducing DSM-supported architecture review to software design education. The study confirmed our previous finding—that it is easy for students to introduce harmful extra dependencies in their implementation—but also explored the reasons behind it: even for students with strong programming skills who invest a lot of effort to ensure modularity, unintentional dependencies were easily introduced and hard to detect without an effective review process. The fact that only instructor-guided review was effective suggests a need for more rigorous architecture review training in software design education.

We are planning another round of experiments at Cal Poly and CMU in the near future. When the process is mature, we intend to extend our investigations beyond the simple application of design patterns. We will explore having students create designs in a more open-ended way, given only a loose description of a problem. Those designs will then be evaluated using architectural reviews, to determine whether they effectively support the kind of software evolution implicit in the description of the problem.

Acknowledgements

This work was supported in part by National Science Foundation under grant TUES-1140752.

References

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

[2] L Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012.

[3] T. Cai, D. Iannuzii, and S. Wong, "Leveraging Design Structure Matrices in Software Design Education". In *Proceedings of the 24th IEEE Conference on Software Engineering Education and Training.* May 2011.

[4] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.

[5] W. Cunningham, "The wycash portfolio management system" In Addendum to the *Proc. Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, pages 29–30, New York, NY, USA, 1992.

[6] E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, Oct. 2004.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.

[9] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software, Architecture", *IEEE Software*, 13:6, Nov. 1996, pp. 47-55.

[10] R. Kazman, M. Barbacci, M. Klein, S. Carriere, and S. Woods, "Experience with Performing Architecture Tradeoff Analysis", *Proc. International Conference on Software Engineering*, May 1999, pp. 54-63.

[11] S. Huynh, Y. Cai, Y. Song, and K. Sullivan, *Automatic modularity conformance checking.* In *Proc. 30th International Conference on Software Engineering*, May 2008, pp. 411–420.

[12] M. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, "Analyzing the evolution of large software systems using design structure matrices and design rule theory". In *Proc. 7th Working IEEE/IFIP International Conference on Software Architecture,* February 2008, pp. 83–92.

[13] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 2000.

[14] D. Parnas, "On the criteria to be used in decomposing systems into modules". *Communications of the ACM*, 15(12), December 1972, pp. 1053–58.

[15] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations". In *Proceedings of the 33th International Conference on Software Engineering* (*ICSE 2011*). Honolulu, Hawaii, USA, May 2011.

[16] L.A. Sudol and C. Jaspan. "Analyzing the Strength of Undergraduate Misconceptions in Software Engineering". In *International Computing Education Research*, July 2010.

[17] L.W. Anderson and D. R. Krathwohl. A taxonomy for learning, teaching, and understanding: A revision of Bloom's Taxonomy of Educational Objectives. Longmans 2001.