Leveraging Design Structure Matrices in Software Design Education

Yuanfang Cai, Daniel Iannuzzi, and Sunny Wong
Dept. of Computer Science
Drexel University
Philadelphia, PA, USA
{yfcai, dli26, sunny}@cs.drexel.edu

Abstract

Important software design concepts, such as information hiding and separation of concerns, are often conveyed to students informally. The modularity and hence maintainability of student software is difficult to assess. In this paper, we report our study of using design structure matrix *(DSM) to assess the modularity of student software by comparing the differences between the DSM representing the intended design and the DSMs representing the software implemented by the students. We applied this approach to a software design class at Drexel University. We found that even though the lab and homework assignments were of small scale, and in many cases, detailed designs were given to the students in the form of UML class diagrams, 74% of the 85 student submissions, although fulfilled the required functionality, introduced unexpected dependencies so that the modules that designed to be independent are actually coupled. These design problems can only be revealed during software evolution, which is usually not possible for student projects. The results show the necessity and benefits of applying DSM modeling to make such design problems explicit to the students.*

1. Introduction

Software design education often focuses on the fulfillment of requirements and properties, such as performance and reliability. Software modularity, though recognized as one of the most important aspects of software quality, is often taught without rigorous analysis techniques. Important design concepts (e.g. information hiding [10], separation of concerns) are conveyed as informal principles. Student assignments are often tested for functionality, but we lack a method to assess modularization decisions that could cause future maintenance problems. As a result, software engineering students often ignore maintainability problems because their homework ends with their class. This deficiency in education contributes to the fact that new software engineers in industry do not pay enough attention to, or do not know how to achieve, maintainability of software—leading to expensive maintenance costs.

In particular, as an important means of achieving modularity in object-oriented programming, design patterns [6] have been widely taught in classroom. Instructors often assess the correctness of a student's design pattern implementation by checking if the corresponding roles and participants are implemented correctly. Moreover, even when a student correctly implements a pattern, she/he may unexpectedly introduce other design problems (e.g. unexpected dependencies [7], bad code smells [4]) that offset the benefits of the pattern, degrade the level of modularity, and incur future maintenance costs. Unlike the functionality of software that can be tested using traditional verification techniques (e.g. unit testing), these modularity problems may not influence software functionality and cannot be easily detected through testing. In addition, two different designs that correctly realize a system's functionality may significantly differ in their maintainability. Traditional design metrics (e.g. coupling and cohesion) may not be sufficient to quantify how well the designs support ease of maintenance. Without a rigorous method to show the differences, it is difficult to convey to students why one design is superior to another in terms of maintainability.

To address these problems, we propose introducing the *design structure matrix* (DSM) [1] model into the classroom of software design education—leveraging the research results of our prior work. As an emerging dependency model, DSMs have been used to visualize the modular structure of software design [12, 1], to compare design alternatives [8, 13], and to identify problematic design decisions and unexpected dependencies [7, 15]. Our idea is to apply these techniques to the assessment of student software. Concretely, given a student software (e.g. an application that applies design patterns), we use DSMs to represent both the *intended* modular structure (called the *model DSM*) and the student's *implemented* structure (called the *sample DSM*). By applying our previously presented *design rule hierarchy* (DRH) algorithm upon the DSMs [16], we can manifest the most important interfaces and how concerns are separate (i.e., which parts of the system should be independent from each other). Comparing the model and sample DSMs, we can identify modularity problems that may cause issues during software evolution.

To evaluate our approach, we applied the approach to the homework and lab assignments of a software design course at Drexel University. The first author routinely instructs an introductory software design course and we applied our approach to the student submissions from a recent course offering. The main objective of this class is to teach students how to apply design patterns in software applications in order to improve software modularity and maintainability. For a subset of labs and homework assignments, we derived the model DSMs from either given UML class diagrams or solutions provided, and verified, by the instructor. We also derived sample DSMs from 85 student submissions. By comparing the samples against model DSMs, we are able to identify that 74% of all submissions have design problems that cannot be readily revealed through testing. Considering that all these labs and assignments are only small scale software systems, and parts of them just implement a given design, the percentage is quite high—showing that modularity problems are extremely common even for small-scale, student projects.

The rest of the paper is arranged as follows: Section 2 illustrates our approach and presents the required background knowledge. Section 3 presents our method of evaluation and results. Section 4 discusses the results and threats. Section 5 describes related work and Section 6 concludes.

2. Approach Overview

In this section, we use a running example to illustrate our approach. Suppose that students are given an assignment to implement a maze game application as designed in Figure 1. This example is a variation of the maze game from Gamma et al.'s [6] book. A maze has a set of rooms; a room has four neighbors, a wall or a door to another room. `MapSite` is the base class for the components of a maze (e.g., `Wall`, `Door`). The UML class diagram shows that the students should apply the *abstract factory*, *builder* and *command* patterns to design a maze game that has the following features. The *abstract factory* pattern supports two alternatives of the maze game: 1. a blue maze game (`BlueMazeFactory`) with a blue wall, black door, and green room; 2. a red maze game (`RedMazeFactory`) with a red wall, brown door, and red room. The *command* pattern allows the user to undo any movements within the maze. The *command* pattern provides two interfaces, a regular command (`Command`) and an undoable command (`Undoable`). A command to move within the maze is handled by a concrete command class `MazeMoveCommand`. Finally, the *builder* pattern facilitates the building of different types of mazes. `MazeBuilder` is the interface to `ConcreteMazeBuilder`, which constructs a blue or red maze by accepting the corresponding factory object as a parameter.

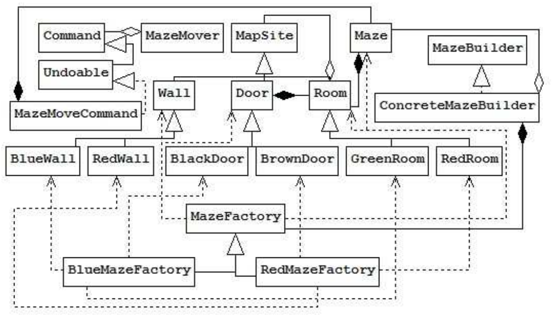
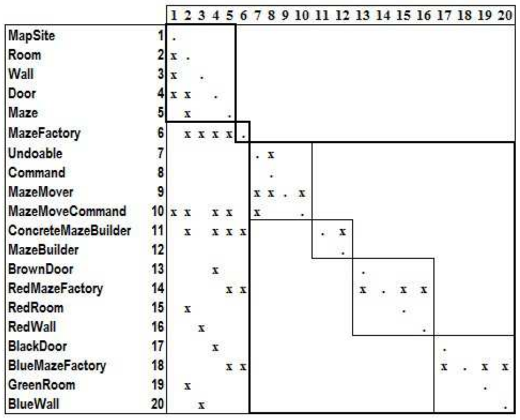


Figure 1. UML of MazeGame

The input of our approach has two parts: the intended design given to the students, and the student software that implements the given design. We first convert both the intended design and student software into design structure matrices (DSMs). The authors have formalized and automated the conversion from both UML class diagram and Java code to DSMs. We also implemented an algorithm that clusters a DSM into a special layered structure called the *design rule hierarchy* (DRH) [16]. After that, we compared whether the student's implementation conformed to the intended design. Next we introduce the concepts of DSM and DRH as background information.

Background. A *design structure matrix* (DSM), as shown in Figure 2, is a square matrix in which the columns and rows are labeled with the same set of variables in the same order. In general, a variable represents a dimension for which a design decision is needed. An "X" in the cell of row i, column j indicates the design decision at row i depends on the decision at column j. In Figure 2, each row/column represents a class shown in the UML class diagram, modeling the fact that each class is a dimension where a decision is needed about its detailed design. The marked cell in row 13 and column 4 indicates that the class `BrownDoor` depends on the class `Door`, derived from the UML in which BrownDoor is a subclass of Door.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
MapSite	1	.																			
Room	2	x	.																		
Wall	3	x		.																	
Door	4	x	x		.																
Maze	5		x			.															
MazeFactory	6		x	x	x	x	.														
Undoable	7							.	x												
Command	8								.												
MazeMover	9							x	x	.	x										
MazeMoveCommand	10	x	x		x	x		x			.										
ConcreteMazeBuilder	11		x		x	x	x					.	x								
MazeBuilder	12												.								
BrownDoor	13				x									.							
RedMazeFactory	14					x	x							x	.	x	x				
RedRoom	15		x													.					
RedWall	16			x													.				
BlackDoor	17				x													.			
BlueMazeFactory	18					x	x											x	.	x	x
GreenRoom	19		x																	.	
BlueWall	20			x																	.

Figure 2. MazeGame DSM

We cluster (i.e., order and group the rows/columns of) each DSM into a special clustering called the *design rule hierarchy* (DRH) [16]. In a DRH-clustered DSM, the design variables are clustered into layers. The first layer contains the most influential decisions that dominate the rest of the system. For example, the first layer in Figure 2 contains the base classes in the UML class diagram that influence the rest of the system. Each layer (the outermost set of squares in the matrix) depends only on layers to the left of it in the DSM. For example, in Figure 2, the second layer only has `MazeFactory`, which depends on the first layer. The third layer in turn depends on the first two layers. Within each layer (squares within squares) the inner squares indicate *independent modules*. For example, the DSM shows that the implementation of red maze game (variables 13–16), and blue maze game (variables 17–20) are completely independent from each other.

Identify Design Problems by Comparing DSMs. A DRH view of a DSM reveals how different concerns are separated into modules, that is, blocks along the diagonal. For example, Figure 2 shows how the system is modularized by the application of the three design patterns. The `MazeFactory` interface in the second layer decouples the design of red maze game (variables 13–16) and blue maze game (variables 17–20). Each factory, `RedMazeFactory` and `BlueMazeFactory`, shows up in its own module with each being dependent on the interface, `MazeFactory`. It even separates the classes involved in the *builder* pattern (variables 11–12) into its own module in the DSM. We simply have a builder interface and a concrete builder dependant on it. The *command* pattern is also separated into a module (variables 7–10). The `MazeMoveCommand` depends on `Command`, which only depends on the base classes contained in the first layer. This DSM shows that such a maze game designed using the three patterns should have at least 6 modules contained in three layers: the base class module, the abstract factory interface module, the command pattern module, the builder pattern module, and two modules each representing one type of maze game.

Our approach is different from other approaches of comparing a design model with an implementation, such as the Reflexion model [9]. These approaches detect discrepancies at a lower-level of granularity. In contrast, our approach only compares the number of modules and the dependency relation *among* modules. The rationale is that it is important to maintain the level of modularity and keep modules independent from each other, and it is also important to allow variation of decisions *within* a module. Our prior work [7] proposed a genetic algorithm to detect the discrepancies between a model DSM, which is derived from a higher-level, more abstract design model, and a sample DSM that is derived from the code that implements the design. The approach we use in this study is similar: we first converted the intended design into a DRH-clustered DSM. This conversion can be done either from a UML class diagram or from implemented code that fulfills the intended design faithfully. We use the DRH-clustered DSM as the *model DSM*. Given a student submission that implements the design, we also derived a *sample DSM* from the source code and clustered it using a DRH. After that, we compared the two DSMs to identify design problems.

For example, for any given student software that is supposed to implement the design shown in Figure 1, we expect to see at least six modules and three layers. If the student's sample DSM showed fewer numbers of modules or the implemented modules do not separate the system as shown in the model DSM, it is obvious that the student did not implement the patterns correctly. Since all the DSMs we study are small-scale projects, the differences can be immediately discerned without using automation tools. If the detailed design is not given, then the instructor or teaching assistant's implementation may not be the best, or there could be multiple designs with the same level of modularity. In this case, we just compare two DSMs to see which one has fewer number of dependencies *between modules*, fewer number of layers, and more independent modules [13].

3. Evaluation

To assess how well the proposed approach can help with software design education by making modularity problems explicit, we evaluated our approach against the following questions:

Q1. Using this method, given an intended design and a student submission implementing the design, is it possible to assess whether the student implemented the intended modular structure correctly? Our purpose is not to check whether the student implemented the required functionality, nor assess the details of how a design pattern is applied. Instead, we assess whether the designed *modular structure* is faithfully implemented. For example, the same pattern can be implemented slightly differently, resulting in a sample DSM that is different from the model DSM only *within* modules. We do not consider these cases as violations. In contrast, if the implementation has fewer modules than designed, it means that at least two modules that were supposed to be separated were not actually separated. These violations are usually caused by unexpected dependencies that hinder software modularity and hence future evolution.

Q2. Is it possible to assess how well modularized a student's submission is? In the cases where a student is only given partial or no UML class diagrams, the student needs to make decisions to complete the rest of the system. The instructor or the teaching assistant can implement a well-modularized solution, which is then converted into a DRH-clustered model DSM. From this model DSM, we can tell how many layers and modules are expected in a well-modularized design, against which a student submission can be assessed.

Q3. For those students who have implemented the same assignment correctly but with different designs, is it possible to assess which design is better modularized? It is possible that a student's implementation is different from, but better than the one given by the instructor or the teach assistant. In this case, we assess the two DSMs against the modularity metrics proposed in Sethi et al.'s [13] work: the better design should have fewer number of dependencies, fewer number of layers and more modules in the last layer.

3.1. Subjects and Evaluation Procedure

To answer these questions, we used labs and homeworks from one introductory software design course. The objectives of this course were to convey important design principles, such as information hiding and open/close principle, and to design software for the ease of maintenance, using Gamma et al.'s [6] design pattern book. This ten-week course included seven labs and two homework assignments. Four of the seven labs required students to implement a given design modeled using UML class diagrams. Since the other three pertained to identifying design structures within existing software, we did not include them in our study. For the two homework assignments, only major components were designed and given, and the students were supposed to make other decisions to make the software complete. Therefore, our study consists of six different small-scale software applications.

The first three labs were concerned with the evolution of the maze game using multiple design patterns. We call each of these labs MazeGame1, MazeGame2, and MazeGame3. MazeGame1 required students to apply a *factory method* pattern to add a red and blue maze. MazeGame2 is similar to MazeGame1, but the students were asked to use an *abstract factory* pattern instead. MazeGame3 evolved from MazeGame2 in that the movement in the maze game should be undoable, using a *command* pattern. The students were also required to apply a *builder* pattern to make the construction of the maze game more flexible. For each of the labs, students were provided with a detailed UML class diagram. Figure 1 shows the design for MazeGame3.

The fourth lab was to implement a coffee shop ordering system, which we call StarBuzz, varied from an example in Freeman et al.'s [5] book. The system allowed a customer to order different types of coffee or tea of different sizes. The system calculated the cost of each order and printed out the receipt of the drink ordered. The cost calculation of a beverage should be implemented using a *decorator* pattern, so that the cost of ingredients of a beverage, such as double moka, can be added and calculated dynamically. The students should have also applied two *strategy* patterns: one to calculate the cost of different sizes and the other to distinguish recipes for tea-based drinks from coffee-based drinks. Similar to the maze game labs, a detailed UML class diagram was provided to the students showing how the *decorator* pattern and two *strategy* patterns should work together.

We refer to the first homework as SurveyManager, which required students to implement a system that allowed the creating, modifying, taking, and automatic grading of surveys/tests. The system was to include six types of questions: true/false, short answer, multiple choice, ranking, matching, and essay. SurveyManager was designed for the students to practice basic object-oriented techniques, such as designing a class hierarchy. In particular, the students were instructed in class that the two concerns, questions in a survey and the responses/answers to the survey should be completely separated, using two separate `Question` and `Answer` interfaces. A `Survey` class that consists of a collection of `Question` type objects, as well as a question class hierarchy were given to the students in the form of a UML class diagram. The students were responsible to design the rest of the system. The students were also told that their design should include a `Grader` class that links/grades a survey/test with its responses/answers. The teaching assistant of the class implemented a standard solution that fulfilled all the requirements.

The second homework, PizzaShop, also varied from a similar example in Freeman et al.'s book, required students to create an application that can be used for ordering pizzas at two store locations, New York and Chicago. The type of crust, sauce, and cheese used varied by location. The students were asked to use an *abstract factory* pattern to accommodate this variation. Additionally, there were several types of pizzas and various toppings that could be applied. The desired solution would include the combination of the following patterns: *abstract factory*, *decorator*, and *strategy*. Similar to the SurveyManager homework, only the structure of the key components was given, and a standard solution was implemented against which the students' submissions were assessed.

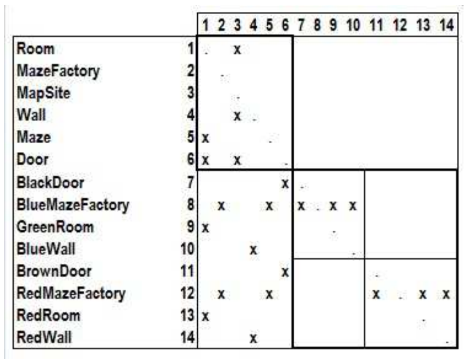
The class had 39 third-year undergraduate students in total. These students all had adequate prerequisites of programming knowledge and skills. We divided the 39 students into 10 groups and each group was supposed to do the labs collaboratively. All the homework was supposed to be done individually. As a result, we have received 40 lab and 78 homework submissions in total. Of all these submissions, we only select as our study subject, the submissions that passed functionality testing. We ended up having 85 usable subjects, 35 of which were labs and 50 of which were homework. Given all the assignments and submissions of the class, we first derived the model DSMs. For the labs, we used the detailed UML class diagrams given to the students as the intended design, and converted the UML diagrams into model DSMs. For the two homework assignments, we converted the standard solution provided by the teaching assistant into a model DSM. We also generated a DRH-clustered DSM from each student submission as a sample DSM. Given that these DSMs are of small scales, we were able to detect the differences between model and sample DSMs manually.

3.2. Results

We summarize the results as follows: of all the 35 lab assignments where detailed UML designs were given, 29% of them did not conform to the given design. For the SurveyManager homework where the question hierarchy and the `Survey` class using that hierarchy were given, as high as 84% of the students were not able to implement the hierarchy and its client correctly. For the parts of the SurveyManager homework and the PizzaShop homework where no detailed design were given, 78% of the class was not able to achieve the same level of modularity demonstrated in the model DSM converted from the standard solution. We analyzed the details of the results as follows.

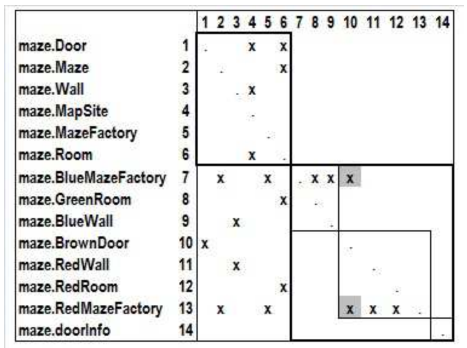
Q1. Assessing the implementation of a given design. It is surprising how likely unexpected dependencies can be introduced by students so that a given design deviates even for a very simple system. Concretely, within the first three labs that are the smallest and most straightforward, only MazeGame3, which was extended from the first two labs, was implemented correctly by all 10 groups. None of the groups implemented the given modular structure in the StarBuzz lab. Only 16% of the students were able to implement the given question hierarchy and `Survey` class correctly in their SurveyManager homework. We now pick a few examples to show the types of unexpected dependencies identified.

Figure 3(b) shows one student's submission of MazeGame2 lab, for which the model DSM is shown in Figure 3(a). Throughout this paper, the grey highlight indicates the modules/dependencies being discussed. At first glance, it appears that the student implemented the given structure correctly because we see the same number of layers and modules in the sample DSM as in the model DSM. However, after further investigation, we notice that both the blue and red modules depend on the



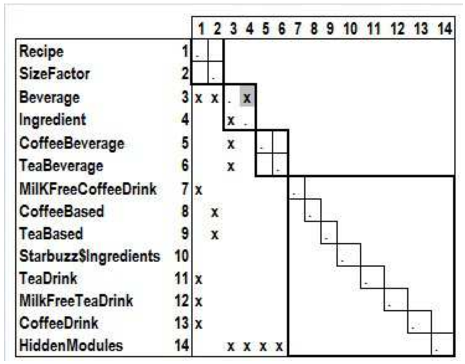
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Room	1	.		x											
MazeFactory	2		.												
MapSite	3			.											
Wall	4			x	.										
Maze	5	x				.									
Door	6	x		x			.								
BlackDoor	7						x	.							
BlueMazeFactory	8		x			x		x	.	x	x				
GreenRoom	9	x								.					
BlueWall	10				x						.				
BrownDoor	11						x					.			
RedMazeFactory	12		x			x						x	.	x	x
RedRoom	13	x												.	
RedWall	14				x										.

(a) Model MazeGame2



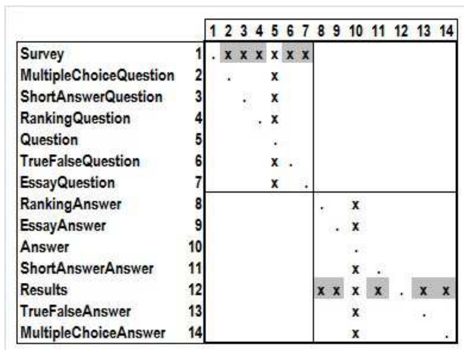
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
maze.Door	1	.			x		x								
maze.Maze	2		.				x								
maze.Wall	3			.	x										
maze.MapSite	4				.										
maze.MazeFactory	5					.									
maze.Room	6				x		.								
maze.BlueMazeFactory	7		x			x		.	x	x	x				
maze.GreenRoom	8						x		.						
maze.BlueWall	9			x						.					
maze.BrownDoor	10	x									.				
maze.RedWall	11			x								.			
maze.RedRoom	12						x						.		
maze.RedMazeFactory	13		x			x					x	x	x	.	
maze.doorInfo	14														.

(b) Sample MazeGame2



		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Recipe	1	.													
SizeFactor	2		.												
Beverage	3	x	x	.	x										
Ingredient	4			x	.										
CoffeeBeverage	5			x		.									
TeaBeverage	6			x			.								
MilKFreeCoffeeDrink	7	x						.							
CoffeeBased	8		x						.						
TeaBased	9		x							.					
Starbuzz$Ingredients	10										.				
TeaDrink	11	x										.			
MilkFreeTeaDrink	12	x											.		
CoffeeDrink	13	x												.	
HiddenModules	14			x	x	x	x								.

(c) Sample StarBuzz



		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Survey	1	.	x	x	x	x	x	x							
MultipleChoiceQuestion	2		.			x									
ShortAnswerQuestion	3			.		x									
RankingQuestion	4				.	x									
Question	5					.									
TrueFalseQuestion	6					x	.								
EssayQuestion	7					x		.							
RankingAnswer	8								.		x				
EssayAnswer	9									.	x				
Answer	10										.				
ShortAnswerAnswer	11										x	.			
Results	12								x	x	x	x	.	x	x
TrueFalseAnswer	13										x			.	
MultipleChoiceAnswer	14										x				.

(d) Sample SurveyManager

Figure 3. Intended Design vs. Implemented Design DSMs

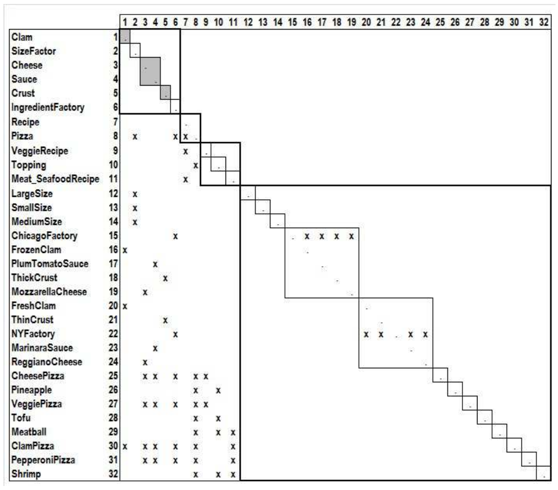
`BrownDoor` class: the student just wanted his blue maze to have a brown door also. This error defeats the purpose of applying the *abstract factory* pattern because the two features of the maze game, blue and red, are no longer independent from each other.

Figure 3(c) shows the reduced DSM of a student's implementation of the StarBuzz lab where the `Ingredient` class is to *decorate* the `Beverage` class and therefore depend on it. However, there should be no dependency on `Ingredient` by `Beverage`, which is identified in this implementation by comparing with the model DSM. Figure 3(d) is a sample DSM of the SurveyManager homework. This design includes the desired `Question` and `Answer` interface, and the answer and question modules are correctly separated. However, the `Survey` and `Results` classes depend on every type of question and answer, respectively. These classes should only depend on the respective `Question` and `Answer` interfaces. This shows that the student did not use polymorphism correctly. This mistake was made by 36% of students; that is, the `Survey` class depends on all concrete question types rather than just the `Question` interface. The results also showed that 28% did not use a `Survey` class and 20% deviated from the intended design substantially. We suspect the reason behind this deviation is that the students had difficulty making the given design work with the rest of their system, so they took the liberty of revising the design to fit their system. All these erroneous dependencies are not detectable by testing and will definitely influence the evolution of the software.

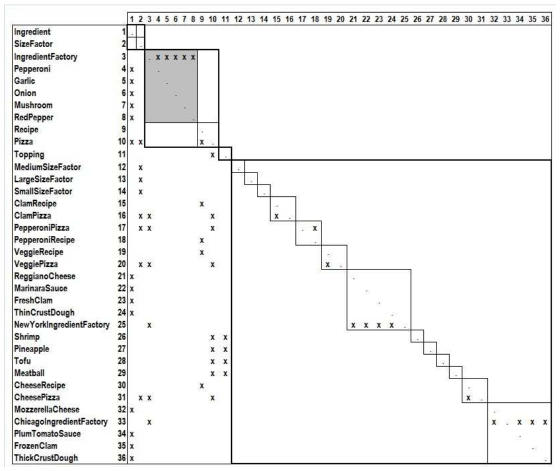
Q2. Recognition of Poor Modularized Structure. Out of 50 homework submissions, for which part or no design was provided, 78% of student submissions did not reach the level of modularization shown in the model DSMs. In the model DSM of the SurveyManager, we observed three layers with 1, 2, and 3 modules respectively, in which the question and answer concerns are totally separated into two modules. In the model DSM of the PizzaShop, we observed 4 layers containing 5, 1, 3, and 15 modules respectively. 78% of the submission contained fewer or erroneous modules, as exemplified below.

In the SurveyManager assignment, one of the most common problem is that students often failed to include an Answer hierarchy. Instead, most students designed the Question class with an Answer object as an attribute. In the model DSM, we can clearly see that a Question module is totally independent of the Answer module (not shown in this paper). Most students' sample DSMs do not have these two concerns completely separated into two modules. For the PizzaShop homework, many students struggled with the factory implementation of the location specific ingredients. Implementations varied significantly from lack of an *abstract factory* pattern to the use of the pattern with extra dependencies on non-shop specific ingredients.

Q3. Comparing Design Alternatives. Of the 85 submissions, only 26% were properly implemented. Once again, more of the homework assignments were improperly done compared to the labs. Students struggled most with the survey homework, as none of the implementations were properly done. Meanwhile, students had more success with the pizza homework, as 44% implemented the correct patterns and had a good architectural design. We compare two such implementations in Figure 4. According to the metrics specified by Sethi et al. [13], the DSM with fewer dependencies, fewer layers and more modules in the last layer is a better modularized design.



(a) Student 1 Sample PizzaShop



(b) Student 2 Sample PizzaShop

Figure 4. Comparison of Valid Implementations

The rationale is simple. The fewer the dependencies, especially inter-module dependencies, the lower the coupling. The fewer the layers, the less the system is vertically constrained and more parts of the system can be implemented and evolved independently. The last layer of the DRH-clustered DSM contains *truly* independent modules, meaning each of these modules can be replaced with a better version without influencing the rest of the system. Accordingly, the larger part of the system falls in the last layer, the better the system is modularized.

Both implementations have four layers and the same number of modules in the last layer. We next have to consider the percentage of the design that falls in the last layer. Figure 4(b) contains 4% more of the design dimensions in the last layer (69% compared to 65%). We also compare the number of dependencies between layers, which is 45 for Figure 4(b) and 50 for Figure 4(a). Based on these observations, we conclude that the design in Figure 4(b) is better modularized. It turns out that the student applied another *factory method* pattern to accommodate the different types of extra ingredients, which improved the design shown in Figure 4(a) slightly. The key differences between the two samples in Figure 4 are denoted by shaded regions in the figure. Figure 4(b) has an `IngredientFactory` module that is not contained in Figure 4(a); while Figure 4(a) contains the `Clam`, `Cheese` and `Sauce`, and `Crust` modules, which are missing from Figure 4(b). The ideally modularized model DSM should contain all of these four modules. We consider both students implemented the design correctly because neither concerns major design components.

4. Discussion

Our results show that even for small scale systems, the likelihood of introducing erroneous dependencies are high. When multiple patterns are applied collaboratively, the students are more likely to introduce unexpected dependencies. Most of the discrepancies we identified in this paper can not be revealed through testing, are not categorized as bad smells [4], but will definitely influence the future maintenance of the software. This study thus shows the necessity and feasibility of applying more rigorous methods, such as DSM modeling, to reveal design problems early and explicitly to the students, improving the education of software design.

In this paper, we identify design problems by manual comparison. For larger systems, we can use existing automated conformance checking techniques [7] to achieve this purpose. In this study, our purpose is to check *modularity violations*, but not to check the correct implementation of design patterns in detail. It is possible that a student implemented the structure of the pattern correctly, but made other mistakes in the detailed implementation of the pattern. On the other hand, not all patterns are for the purpose of improving modularity, such as the *singleton* and *flyweight* patterns. The lab and homework assignments used in this study do not cover all the patterns.

Different from our prior work [16] that applied DRH clustering to *augmented constraint network* (ACN) [2] models for the purpose of identifying parallel tasks, in this paper, we applied DRH clustering to DSM models directly derived from UML models (without using ACNs) to capture modules that are separated by architecture decisions, such as the application of design patterns. Our previous work showed that ACN modeling can capture implicit and indirect dependencies that are not explicitly visible in UML diagrams. Our future work is to investigate if we can obtain different results using ACN-derived DSMs. The metrics we applied in this study are limited. We plan to introduce and apply more metrics, based on DSM modeling, into software modularity education.

5. Related Work

Preiss [11] presents the idea that design patterns should be integrated early into CS curricula, arguing that design patterns are a key part in constructing well designed software systems. In this paper, we use DSMs to visualize the significant impact of design patterns on software design structures, against which the students' application of design patterns, in the regards of modularity improvement, can be rigorously assessed. Moreover, our approach also assesses students' ability of making other design decisions, such as the basic class hierarchy design.

Sobel and Campbell [14] introduced Advanced Design Employing Pattern Templates (ADEPT) that incorporates proper use of design patterns in student designs as part of their formal analysis. Sobel and Campbell claim that formal analysis education leads to a higher ability to construct well-designed software systems. Our work aims to ensure the intended modularity. For example, we found that even if a design pattern is accurately applied, the students can introduce unexpected dependencies that offset the effects of the design pattern.

There are several papers published on conformance checking that are directly related to our comparison of two DSMs. Murphy et al. [9] proposed the software reflexion model to structurally compare a high-level model against a model derived from the source code. Christl et al. [3] extend Murphy et al.'s work in attempt to semi-automate the conformance checking of the high-level design and source in the reflexion method using clustering techniques. By contrast, our work checks conformance at the level of *modules*.

6. Conclusion

In this paper, we reported our study of applying design structure matrix modeling to the education of software design. In particular, we showed how to explicitly identify modularity problems in students' software by comparing the model DSM that represents the intended design with sample DSMs that represent students' implementations. We found that even for small scale systems where the intended design is given, a student is still very likely to introduce unexpected dependencies that degraded the intended modularity. This study shows the benefits and necessity of introducing DSM modeling into software engineering education.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

References

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

[2] Y. Cai and K. J. Sullivan. Simon: A tool for logical design space modeling and analysis. In *Proc. 20th ASE*, pages 329–332, Nov. 2005.

[3] A. Christl, R. Koschke, and M.-A. Storey. Equipping the Reflexion model with automated clustering. In *Proc. 12th WCRE*, pages 89–98, Nov. 2005.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[5] E. T. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, Oct. 2004.

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

[7] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proc. 30th ICSE*, pages 411–420, May 2008.

[8] 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 83–92, Feb. 2008.

[9] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. 3rd FSE*, pages 18–28, Oct. 1995.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–8, Dec. 1972.

[11] B. R. Preiss. Design patterns for the data structures and algorithms course. In *Proc. 30th SIGCSE*, pages 95–99, 1999.

[12] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th OOPSLA*, pages 167–176, Oct. 2005.

[13] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. Joint 8th WICSA and 3rd ECSA*, pages 269–272, Sept. 2009.

[14] A. E. Sobel and S. Campbell. Supporting the formal analysis of software designs. In *Proc. 20th CSEET*, pages 123–132, July 2007.

[15] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd ICSE*, Nov. 2011.

[16] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th ASE*, pages 197–208, Nov. 2009.