# Simulating Structural Design Evolution of Software

Warren Baelen
Drexel University
Philadelphia, PA, USA
warren.s.baelen@drexel.edu

Yuanfang Cai
Drexel University
Philadelphia, PA, USA
yfcai@cs.drexel.edu

## ABSTRACT

Software designers frequently find the need to refactor existing code to improve the structure of their design. Typical examples include the application of design patterns or splitting large modules into smaller ones. The problem lies in the difficulty in evaluating the effect of such restructuring before developers implement their newly proposed design ideas. Such experimental implementation can be expensive because it may include potential conflicts or cause unnecessary source code branching to occur. It can be even more expensive if the designer needs to compare multiple design alternatives. In this paper, we propose to demonstrate the possibility of simulating structural design changes based on Baldwin and Clark's modular operators. This simulation permits designers to explore the design space and validate ideas for software changes without having to implement them concretely.

## Keywords

design rule theory, modularity assessment

## Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*design modeling*

## 1. INTRODUCTION

Designers must continually assess and improve the quality of their proposed changes while other developers make underlying changes of their own. The necessity of frequently refactoring an existing codebase to maintain a good design structure has been recognized. The most representative activity of such refactoring is splitting a large module into smaller ones by, for example, applying a design pattern. The results of these changes are hard to envision or evaluate prior to actual implementation which can be problematic. It is very possible that the change overlaps or impacts the current work for other implementers. It is also often the case that there are more than one possible refactoring strategy and evaluating which one would be more effective can be difficult to assess before both methods are implemented.

There are several tools which utilize Unified Modeling Language (UML) diagrams to understand system interactions and changes [7]. The tools themselves tend to be fine grained and allow users to modify classes and dependencies in an ad-hoc way, without showing the overall variation of the modular structure before and after refactoring [2]. There are also tools on the market that are dedicated to manipulating the dependency structure of code to decrease complexity via dependency analysis and simulation, such as Lattix [5]. Although these tools can be used to conduct fine-grained simulation, more problems remain.

First, these tools assume that *modules* are always created by organizing the code into packages or namespaces, That is, packages/namespaces are considered the dominating way of decomposing *modules*. However, a *module* should be an independently evolvable unit [1,6], which may or may not be coincident with the package or namespace structure. These tools are not designed to review true *modules* that are mutually independent, hence the simulation supported are not at the level of *true modules*.

Second, these tools usually define *design rules* in the form that code within a module cannot depend on code within some other modules. Such design rules maybe violated when developers simply mislabel a class into an unintended package. These tools allow the user to simulate dependency addition/removal between individual files to remove such violations. The problem is that such simulation are not designed to reveal the overall variation of the modular structure. Concretely, they do not show if the number of independently evolvable modules are increased or decreased.

In addition, the most typical refactoring activity is to add interfaces or design patterns to split large modules into smaller ones. According to Baldwin and Clark, such interfaces are also called *design rules*, representing stable and architectural level decisions. Inserting design rules is the key activity to create true modules. Existing tools are not designed to simulate such module-level operations.

In this paper, we propose to automate a subset of the six modular operators defined by Baldwin and Clark [1], *splitting*, *substitution*, *augmenting*, *removing*, *inverting* and *porting*. These operators work on *modules* and *design rules*, emphasizing the stability of architectural decisions and the independence among modules. Our purpose is to allow designers to simulate and evaluate envisioned changes and refactoring proposals in terms of their impact on overall design structure before they actually implement these changes.

Our tool is based on an *augmented constraint network* which formalizes the concept of *design rule* and *modules*. The modular structure of a design will be visualized using *design structure matrices* (DSM) that can be automatically generated. The true modules will be reviewed by clustering a DSM using the *design rule hierarchy*. In Section 2, we will briefly introduce these key concepts. In Section 3 we then show the automated application of the modular operators using a prototype tool called *Metis*. We also demonstrate the feasibility of simulating proposed changes using Metis with a small example.

Since the underlying computational model are the augmented constraint network and design structure matrix, our approach has the potential to simulate envisioned changes both on existing source code, which can be reverse engineered into an ACN, and on design models such as UML class diagram, which also can be automatically transformed into ACN models. Our goal is to provide a lightweight simulation tool to assist with reasoning about structural software design evolution.

## 2. BACKGROUND

*Design Rule Theory and Modular operators.* The theoretical foundation of our work is Baldwin and Clark's design rule theory [1]. The basic idea is that design rules, that is, architectural design decisions, frame the modular structure of a system. Their theory is highly consistent with the concept of *information hiding* [6], which states that design decisions that are likely to change should be hidden behind stable and visible design decisions–abstract interfaces.

Baldwin and Clark used six modular operators to describe the evolution of a design as a process described as *design rationalization*. In this process, designers reason about the structure of their design and create modular structures. This phase also describes subsequent changes to a design once it has become modular. The modular operators are: *Splitting*–a design rule is introduced to split the otherwise coupled decisions into mutually independent modules, *Augmenting*—add a module to a design, *Exclusion*—remove a module from a design, *Substituting*—replace a module with another version, *Inverting*—extract common concerns from different modules and creates a new module with a set of design rules to support the common activity, and *Porting*—move a module from one design space to another.

Once applied to a system, these operators can transform it into a multitude of different structures and the process of rationalization continues as the system grows and takes on new requirements. For the sake of space, in the rest of the paper we mainly elaborate on the most important operator, *Splitting*, that generates true modules that can be manipulated by the other operators.

*Design Structure Matrix and Design Rule Hierarchy.* The Design Structure Matrix (DSM) is a representation for analyzing dependencies between variables. A DSM is an adjacency matrix with design variables repeated along both axes. Where there are dependencies between the two variables an $\times$ will appear and since a variable relation on itself has no meaning, there are set of $\bullet$ characters across the diagonal.

We use DSMs to visualize the two key concepts, design rules and modules. Wong & Cai's *Design Rule Hierarchy* (DRH) algorithm automatically reveals the design rules in a DSM and the modules they frame [9]. The top level of a DRH contains decisions that influence other decisions, but do not depend on any other design decisions below them. The variables in the middle layers of the hierarchy are the design rules of the subsequent layers. Each layer of the hierarchy contains *modules*, that are mutually independent from each other and split by upper level design rules.

*Augmented Constraint Network.* The basic elements of modular operators are design rules and modules, which in turn, are based on pair-wise dependencies among design decisions. In order to rigorously define and reason about dependency relations among design decisions, Cai & Sullivan created a formal dependency model called the *Augmented Constraint Network* (ACN) to express the design space for software projects [2]. An ACN consists of a *constraint network*, a dominance relation, and a set of clusterings.

The constraint network forms the base of an ACN and it contains a set of variables modeling possible design dimension. Each variable has a domain that contains a number of values that model the possible choices of the design dimension. The dependency relation among these decisions are modeled as logical expressions. The dominance relation of an ACN is binary relation among variables, modeling the fact that some design decisions cannot be influenced by subordinating decisions. Each clustering within the clustering set of an ACN models one way these design variables can be partitioned. From an ACN, a DSM with rigorous semantics can be automatically derived. In this paper, we evaluate a tool created with the concept of modular operators based on ACN models.

## 3. TOOL & EVALUATION

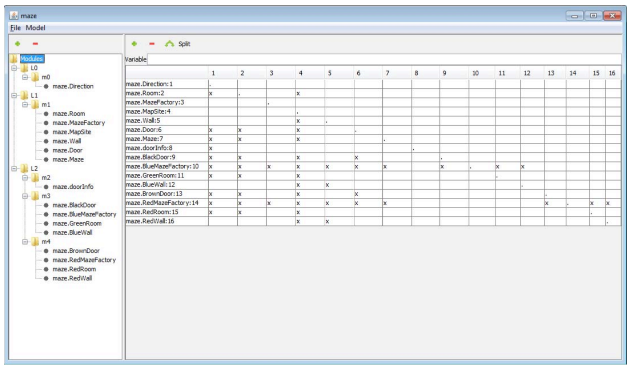
Our prototype tool, called *Metis*, was designed to the determine the feasibility of using modular operators to simulate design evolution.

**Metis System.** Metis takes an ACN model as input, transforms the ACN model into a DSM that can be viewed and manipulated by the user. The ACN model can be generated from either source code or design models using our other tools [9]. Once an ACN model is opened, its Design Structure Matrix (DSM) and corresponding Design Rule Hierarchy (DRH) clustering are created using the Parcae tool [9]. The clustering is then populated in a tree control on the left pane. Users can focus their area of analysis by selecting different parts of the clustering tree. A grid view of the DSM is shown on the right pane and allows users to see the relations of the variables they have already selected. Figure 1 shows a snapshot of the Metis tool.

Metis allows the user to simulate design changes based upon our previous formalization of the modular operators. Designers may load an ACN model and begin to apply the modular operators, such as *Split*, *Augment*, and *Exclude* to the model. They can continue to apply the modular operators, add and exclude singular variables, as well as save and refresh a model as needed.

Figure 2 demonstrates the *Split* operation using Metis. The dependencies to be divided are selected and the "Split" button is pressed. A dialog then appears which allows the user to specify either an existing design rule or specify a newly proposed one to break the relationship between the two variables. Once the design rule is specified, the model will be reprocessed and DSM and DRH both refreshed - revealing new modules.

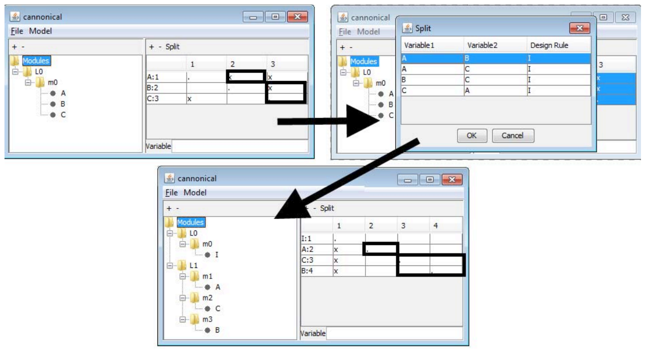
The small system has three design variables, `A`, `B`, and `C` [4, 8]. `B` and `C` form a *proto-module* with `A`. A proto-module is a



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| maze.Direction:1 | . | | | | | | | | | | | | | | | |
| maze.Room:2 | x | . | | x | | | | | | | | | | | | |
| maze.MazeFactory:3 | | | . | | | | | | | | | | | | | |
| maze.MapSite:4 | | | | . | | | | | | | | | | | | |
| maze.Wall:5 | | | | x | . | | | | | | | | | | | |
| maze.Door:6 | x | x | | x | | . | | | | | | | | | | |
| maze.Maze:7 | x | x | | x | | | . | | | | | | | | | |
| maze.doorInfo:8 | x | | | | | | | . | | | | | | | | |
| maze.BlackDoor:9 | x | x | | x | | x | | | . | | | | | | | |
| maze.BlueMazeFactory:10 | x | x | x | x | x | x | x | | x | . | x | x | | | | |
| maze.GreenRoom:11 | x | x | | x | | | | | | | . | | | | | |
| maze.BlueWall:12 | | | | x | x | | | | | | | . | | | | |
| maze.BrownDoor:13 | x | x | | x | | x | | | | | | | . | | | |
| maze.RedMazeFactory:14 | x | x | x | x | x | x | x | | | | | | x | . | x | x |
| maze.RedRoom:15 | x | x | | x | | | | | | | | | | | . | |
| maze.RedWall:16 | | | | x | x | | | | | | | | | | | . |

**Figure 1: The Metis proto-type tool. The clustering of the variables appear in the left pane and the DSM on the right.**

group of variables with dependencies on each other but need to evolve separately. In this case there exists a symmetric dependency between variables `A` and `B` and only `B` has a dependency on `C`. After applying the splitting operator using an interface variable `I`, the two modules (`A` and `B,C`) become independent from each other as they both only depend on design rule `I`.



| | 1 | 2 | 3 |
|---|---|---|---|
| A:1 | . | x | x |
| B:2 | | . | x |
| C:3 | x | | . |

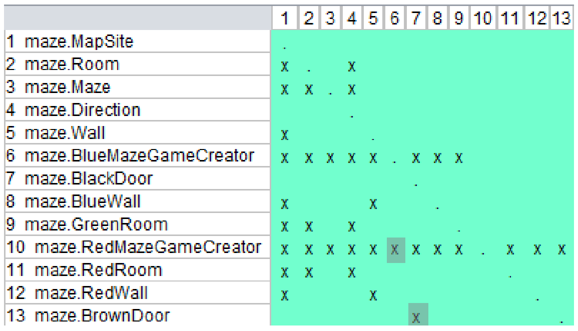
| Variable1 | Variable2 | Design Rule |
|---|---|---|
| A | B | I |
| A | C | I |
| B | C | I |
| C | A | I |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| I:1 | . | | | |
| A:2 | x | . | | |
| C:3 | x | | . | |
| B:4 | x | | | . |

**Figure 2: The Metis tool performing the *Split* operation on variables `A,B` and `C` with the specified design rule `I`**

**Simulating Design Evolution.** Now we use a maze game example modified from [3] to demonstrate the possibility of simulating design evolution and refactoring. This example is about creating a maze game with various elements, such as rooms connected through walls and doors. We simulate the application of *Split* twice to refactor an existing design and the application of *Augmenting* to add a new feature.

Figure 3 depicts a DSM modeling a design created as follows: at the beginning, the system only creates basic maze game with rooms but not doors. Then it evolved to create mazes with black doors and blue walls connecting green rooms. `BlueMazeGameCreator` is the main program that can create both the basic maze or a *blue maze game*. The `GreenRoom` and `BlueWall` extends the `Room` and `Wall`. The system then evolved to include a *red maze game* with brown doors, red rooms and red walls. The `BrownDoor` extends `BlackDoor`, and `RedMazeGameCreator` inherits from `BlueMazeGameCreator`. The DSM was generated from an ACN which can be derived from the source code implementing the above design.

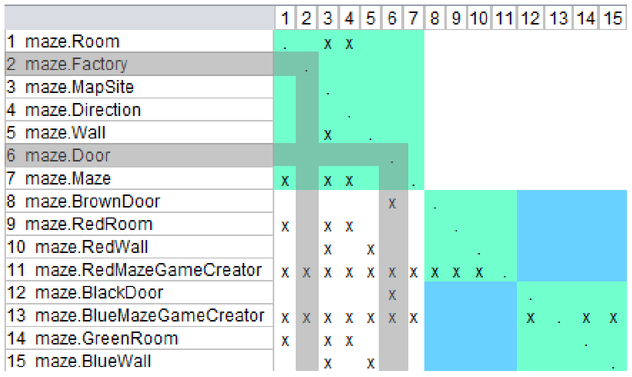


| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 maze.MapSite | . | | | | | | | | | | | | |
| 2 maze.Room | x | . | | x | | | | | | | | | |
| 3 maze.Maze | x | x | . | x | | | | | | | | | |
| 4 maze.Direction | | | | . | | | | | | | | | |
| 5 maze.Wall | x | | | | . | | | | | | | | |
| 6 maze.BlueMazeGameCreator | x | x | x | x | x | . | x | x | x | | | | |
| 7 maze.BlackDoor | | | | | | | . | | | | | | |
| 8 maze.BlueWall | x | | | | x | | | . | | | | | |
| 9 maze.GreenRoom | x | x | | x | | | | | . | | | | |
| 10 maze.RedMazeGameCreator | x | x | x | x | x | x | x | x | x | . | x | x | x |
| 11 maze.RedRoom | x | x | | x | | | | | | | . | | |
| 12 maze.RedWall | x | | | | x | | | | | | | . | |
| 13 maze.BrownDoor | | | | | | | x | | | | | | . |

**Figure 3: The Maze DSM prior to applying *Split* operator. All variables form a single large module, and the highlighted dependencies are candidates for introducing new design rules.**

After applying the DRH clustering, the DSM in Figure 3 shows that there are no mutually independent modules. We box the groups of variables to show the intended proto-modules: the basic maze (top box), blue maze (middle box) and red maze (bottom box). Since the basic maze contains basic classes that can be seen as design rules, we aim to separate the blue and red maze modules.

*Applying Splitting Operator.* We propose two steps to achieve this purpose. First, we plan to abstract the commonality between `BlackDoor` and `BrownDoor` using a `Door` class to decouple the two concrete classes. This can be modeled as a *Split* operator on these two variables using `Door` as a design rule. The second step is apply an abstract factory pattern to separate the creation of two types of maze game. The application of the pattern can be modeled as splitting the two blue and red maze modules by inserting a design rule `maze.Factory`.



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 maze.Room | . | | x | x | | | | | | | | | | | |
| 2 maze.Factory | | . | | | | | | | | | | | | | |
| 3 maze.MapSite | | | . | | | | | | | | | | | | |
| 4 maze.Direction | | | | . | | | | | | | | | | | |
| 5 maze.Wall | | | x | | . | | | | | | | | | | |
| 6 maze.Door | | | | | | . | | | | | | | | | |
| 7 maze.Maze | x | | x | x | | | . | | | | | | | | |
| 8 maze.BrownDoor | | | | | | x | | . | | | | | | | |
| 9 maze.RedRoom | x | | x | x | | | | | . | | | | | | |
| 10 maze.RedWall | | | x | | x | | | | | . | | | | | |
| 11 maze.RedMazeGameCreator | x | x | x | x | x | x | x | x | x | x | . | | | | |
| 12 maze.BlackDoor | | | | | | x | | | | | | . | | | |
| 13 maze.BlueMazeGameCreator | x | x | x | x | x | x | x | | | | | x | . | x | x |
| 14 maze.GreenRoom | x | | x | x | | | | | | | | | | . | |
| 15 maze.BlueWall | | | x | | x | | | | | | | | | | . |

**Figure 4: The Maze DSM after applying *Split* operator, which introduces both the `Door` and `MazeFactory` design rules. This causes two seperate modules to be created.**

*Applying Augmenting and Excluding Operators.* Now we simulate the situation where a new *orange* game needs to be added. In order to visualize the impact of this change without actually implementing it, we can first create an ACN modeling an `OrangeMaze` module. This model can be derived from a UML class diagram showing the intended design. This module contains both the variables needed for the orange maze, it also models external design rules, that is, the `Factory` interface in the abstract factory pattern, according to the definition of *module* from our formalization.

We are then able to add this new type of maze to the larger Maze model simply by selecting the Augment button and selecting a module file. The Maze model is then refreshed and our new module is revealed in the DSM and DR (Figure 5). This has the implication of allowing designers to combine work by several other designers and see how it will impact the overall model. This also raises the question of another feature to export the design rules of a model to assist designers coordinate. In addition the *Exclude* operator was easily performed on our `OrangeMaze` and removed the added model reverting the DSM and DRH to its previous state.

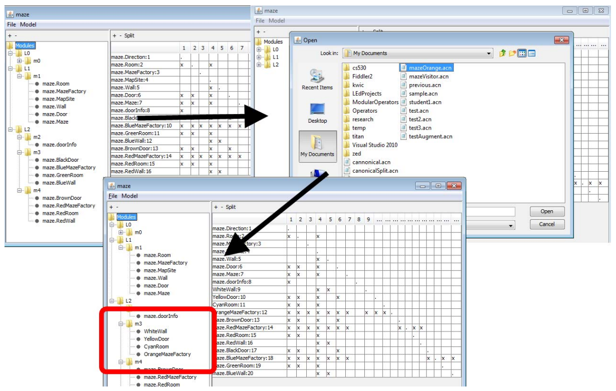


**Figure 5: The Maze model before and after augmenting the `OrangeMaze` which is highlighted in red.**

## 4. CONCLUSION AND FUTURE WORK

In this paper, we demonstrate the feasibility of simulating design evolution and refactoring by automating the use of Baldwin and Clark's modular operators using a prototype tool called *Metis*. This approach has the potential to help in the design rationalization process without forcing designers to realize their ideas concretely in software. Although we have only evaluated a small sample system, we plan to extend our evaluation to larger industrial implementations, and refine the tool to improve scalability and usability. For example, we will extend the tool to support formatting the DSM view to highlight proto-modules. Another possible usability feature would be to extend the Split dialog to apply the same design rule to all relations that contain a variable.

## 5. REFERENCES

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

[2] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis.* PhD thesis, University of Virginia, Aug. 2006.

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

[4] M. J. LaMantia, Y. Cai, and A. D. M. J. Rusnak. Analyzing the evolution of large software systems usingdesign structure matrices and design rule theory. In *Proc. 7th WICSA*, pages 83–92, Feb. 2008.

[5] B. Merkle. Stop the software architecture erosion. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 295–297, New York, NY, USA, 2010. ACM.

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

[7] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Argo: a design environment for evolving software architectures. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 600–601, New York, NY, USA, 1997. ACM.

[8] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. 8th FSE*, pages 99–108, Sept. 2001.

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