Managing Variabilities in On-going Design

Sushil Krishna Bajracharya and Cristina Videira Lopes
Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{sbajrach,lopes}@ics.uci.edu

ABSTRACT

A notion of ephemeral design variants that originate as a result of ongoing design changes is presented. The idea is based on the fact that the side effects of a design-change can be handled in more than one way, and, each possibility to fix the side effect is a design option to be considered. An approach to identify such design variants in a program based on the dependency model of the program is discussed. The applicability of this notion of design variants is demonstrated using an example from refactoring using Dependency Structure Matrix (DSMs) as a possible interaction tool.

1. INTRODUCTION

Software variability management deals with the problems of modification, customization and reconfiguration of software systems during various life cycles. Most of the research in variability management deal with variabilities of the product within the two dimensions; space and time. [4] This line of work does not cover managing the variabilities in design, that exist as ephemeral design options, as the design is constantly changed. Any well designed program has elements with some dependency structure. Making changes in one or more elements will have side effects on its dependents. This makes software design a decision intensive process where design changes present various opportunities for handling the consequences of such changes. For example, in a typical case of refactoring, Fowler presents three possible options in handling the effects of *Extract Class* refactoring ([6], page 149). These options are in fact variabilities in design. Some of these options would otherwise go unnoticed if the designer (or the one making changes) lack proper insights or cannot foresee all the possibilities that result from the ongoing modifications.

In the rest of this paper we present some idea to stir up discussion on making these ephemeral options more tractable. Our approach is based on using dependency models to identify and analyse these options.

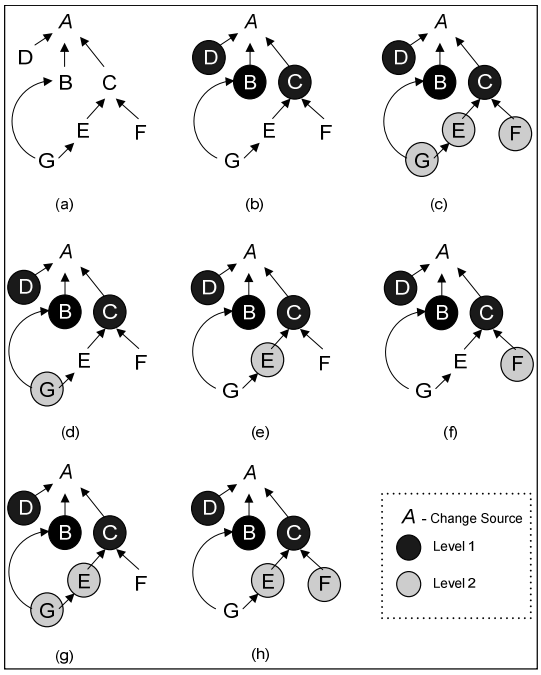

Figure 1: Various options for handling ripple effects

2. VARIATION POINTS AND DEPENDENCY MODELS

An important aspect in managing variabilities is identifying the *Variation Points*, *one or more points at which variations will occur* [15]

Program dependencies modeled as a directed graph is a fundamental structure [7] used pervasively in sofware engineering. The nodes in the dependency graph can be taken as the variation points in our case. A change in a node (or nodes) in the graph will possibly trigger further changes in the dependent nodes. Such a ripple effect that originates from the source and propagates to its dependents can be handled in more than one way. These variations of possible change-fixes are, what we call, *variabilities in the ongoing design.*

The possibilities of variations of such ripple effects are shown in Figure 1. It shows a dependency structure as a directed graph. We can assume the nodes (letters 'A' - 'G') to be modules and the direction of arrow indicating a depencency. Thus, 'B' depends on 'A', 'E' depends on 'C', and

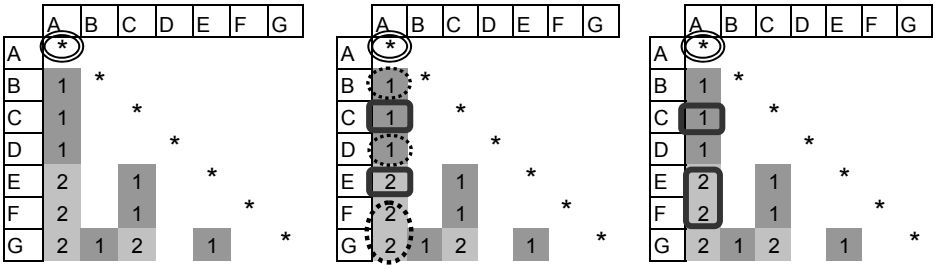

Figure 2: Configuring changes using DSM

so on. Rest of the graphs (b) to (h) show all the possibilities of side effects of a change made in 'A'. The nodes enclosed in circles indicate that they were changed due to the change that originated in module 'A'. Figure 1.(b) shows the case where the changes made in 'A' only affected its direct dependents highlighted by the black circles. Figure 1.(c) presents the case that shows how the change made in 'A' has affected all the nodes, direct (black) and indirect/transitive (grey) dependents. (Node 'E' and 'F' are at second level, 'G' is both at second and third level). Figure 1.(d) to (h) show other possible cases where the effects of changes in 'A' were compensated at various nodes.

Simply generating all the possible configurations (like from Figure 1.(a) to (h)) will not necessarily give the valid list of design variants. Many such configurations could simply be meaningless. The possible options that are valid and that can be considered will primarily be constrained by the the kinds of relationship that hold between the elements in the dependency model. These models can range from simple call graphs to social networks of developers .

A precisely formulated dependency model is a prerequisite for identifying the possible design variants. In order to properly reason about these variants or the options for compensating a design change, we need to be able to; (i) identify these changes, (ii) for each kind of change list the possible valid compensation mechanisms, and (iii) pick the most viable one. There can be several criteria to decide among the variants. For example, change impact analysis [11, 3] based on source code analysis provides several such criteria. Another possibility might be based on program metrics [10] or analysis techniques for modularity [9, 14]. Also important is the formal underpinnings necessary to precisely formulate the dependency models. [5] Further elaborate discussion of all these issues is out of the scope of this paper and are of further research interest.

3. DSMS FOR MANAGING CHANGES

In this section we present Dependency Structure Matrix (DSM) [13, 1] as a viable representation for dependency graphs and as a possible interface for a tool to manage variations in design. Such interface can possibly be used in identifying the variation points (primary nodes where change occur and the dependents of these nodes) and specifying how the variations should be handled.

Figure 2 shows a possible interface for selecting the various possibilities of handling changes using DSMs The DSMs indicate dependencies using numbers that show the level of dependency. The leftmost DSM models the dependency structure shown in Figure 1. It shows all the dependents of module 'A' (entries in the second column) using numbers for the level of dependencies (1 meaning direct, 2 meaning second level dependency and so on). The middle DSM shows a possibility where a designer can formulate a query like; *Can I compensate for change in 'A' (concentric circle) by limiting the side-effects upto modules 'C' and 'E' (solid rectangles), and without making any changes in 'B', 'D', 'F' and 'G' (dotted circles)?* A tool implemented this way would be able to suggest if such compensation option is feasible or propose new ways. For example, by proposing changes as shown in the rightmost DSM which models the change structure shown in Design 1.c, in Figure 1. With such interaction, a designer will be able to identify various change alternatives (such as, Design 2.b Vs Design 2.c).

4. AN EXAMPLE FROM REFACTORING

Let us consider an example in Java as shown in Figure 3. As a possible design change let us consider performing the refactoring *Extract Class* on field `b1`, in class `B`. This would typically include creating a new class, `BField1` (say) and adding an accessor `getBField()` in class `B`. This change made in `B` will have to be appropriately compensated by making fixes in its clients (or dependents).

Figure 4 depicts the DSM for the source given in Figure 3, which shows the full transitive closure of the dependencies in it. This DSM was generated using the analysis results from the tool *Dependency Finder*. Thus, the dependencies shown are dependencies between *Classes* as well as *Features*. [2] This figure illustrates how DSMs can be used as the frontend for identifying and possibly specifying the selection of design variations in refactorings. The DSM shows the following possible interactions;

1. Picking a Module to *Extract Class* from. This is the start of the refactoring.

2. Identifying the sources of changes;

   - Field to be replaced `b1`, and
   - Method to be modified `getB1()`
   - Observing the possible clients, dependents highlighted in the columns
   - Selecting the clients to whom the new introduced class would be exposed
   - Selecting the clients that would not see the new field, and continue using the old interface; `getB1()`

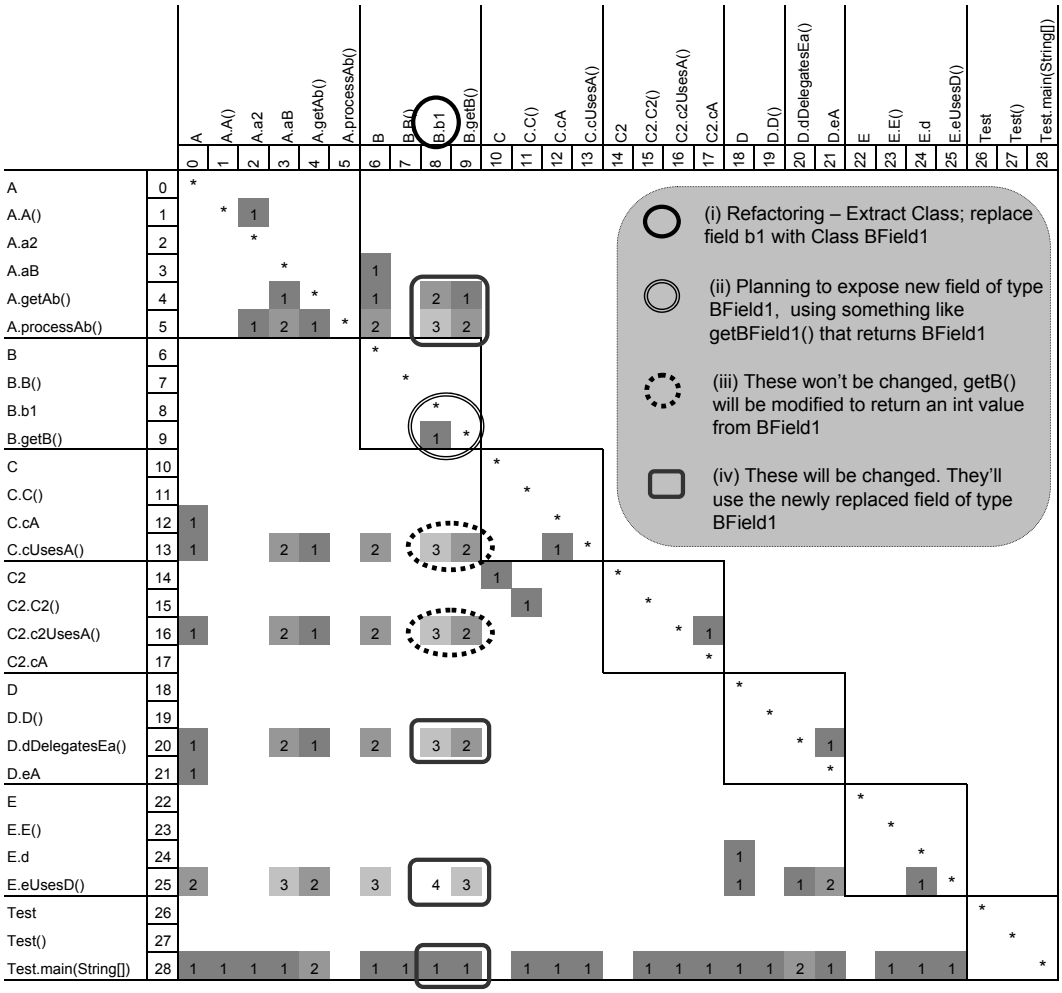

Figure 4: DSM for the code shown in Figure 4

The power of DSMs lie is their ability to show the holistic view of dependencies in a single matrix. This makes it possible to think aboout and iterate through the design variations that would otherwise have gone unnoticed while making design changes.

5. FEASIBILITY

The most important question about the feasibility of using DSM for a purpose like this is how they would scale up to handle large systems. DSMs have been used in design analysis of moderately large scale software. [12] Furthermore, with fine grained refactoring, the DSMs need not show all the nodes in the program. It would be enough to show those elements that are captured by the variability points in the dependency graph.

More crucial is the feasibility of selecting and composing the nodes (as shown in Figure 4). The impact of a given change (refactoring, for example) also depends largely on what the program elements represent which come largely from the Application Domain [8]. These are the research topics that need to be clarified further.

5.1 Tool support for refactoring

Current refactoring tools have very limited support, not beyond small basic refactorings. Two of the popular IDEs we used (Eclipse and IntelliJ IDEA) do not yet support refactorings like *Extract Class*. So the refactoring as shown in the example above, can be performed by manually executing the proper mechanics and composing smaller refactorings as explained in [6]. Additional support by incorporating the idea as we have proposed adds a novel approach in providing tool support for refactoring.

6. CONLCLUSION

We have presented the case of managing variability in design based on the dependency structure of program. We proposed using DSMs as a possible front end to handle such variations. Software variability management does not yet consider the case of managing the variation in on-going design that exist as possible design choices. We believe treating these choices as design variations and providing tool support for managing them will be instrumental in augmenting the capabilities of designers/developers, making them aware

```
class B{
    int b1;
    public int getB(){return b1;}
}

public class A {
    public B aB;
    int a2 = 2;
    public int getAb(){
       return aB.getB();
    }
    public int processAb(){
        return getAb() * a2;
    }
}

class C{
    public A cA;
    public void cUsesA(){
        System.out.println(cA.getAb());
    }
}

class C2 extends C{
    public void c2UsesA(){
        System.out.println("##" + cA.getAb() + "##");
    }
}

class D{
    public A eA;
    public int dDelegatesEa(){
        return eA.getAb();
    }
}

class E{
    public D d;
    public void eUsesD(){
        System.err.println(d.dDelegatesEa());
    }
}
```

Figure 3: Java Code Sample

of and more capable in handling unforeseen design options.

7. REFERENCES

[1] Tutorials and resources on DSM, DSM web site *http://www.dsmweb.org*.

[2] Dependency Finder Web Site *http://depfind.sourceforge.net/*.

[3] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[4] J. Bosch. Software variability management. *Science of Computer Programming*, 53:255–258, December 2004.

[5] Y. Cai and K. J. Sullivan. A value-oriented theory of modularity in design. In *Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research (EDSER) at ICSE'05*, May 2005.

[6] M. Fowler, K. Beck, J. Brant, O. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[7] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM Press.

[8] M. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[9] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[10] K. Maruyama and K. ichi Shima. Automatic method refactoring using weighted dependence graphs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 236–245, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, 2004. ACM Press.

[12] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. *OOPSLA 05*, 2005.

[13] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28:71–74, 1981.

[14] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108. ACM Press, 2001.

[15] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53:305–331, December 2004.