Vrije Universiteit Brussel

Faculteit Wetenschappen
Vakgroep Computerwetenschappen

Design Structure Matrices for Software Development

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Matthias Stevens

Promotor:	Prof. Dr. Theo D'Hondt
Begeleiders:	Dr. Johan Brichau
	Dr. Andy Kellens

20 augustus 2007

Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science

Design Structure Matrices for Software Development

Graduation thesis submitted with intention to obtain the degree of Licenciate in Computer Science

Matthias Stevens

Promotor:	Prof. Dr. Theo D'Hondt
Advisors:	Dr. Johan Brichau
	Dr. Andy Kellens

August 20, 2007

Nederlandstalige Samenvatting

In deze dissertatie presenteren we ons onderzoek naar toepassingen van Design Structure Matrices (DSMs) en gerelateerde technieken in de context van software ontwikkeling. We spitsen ons toe op twee invalshoeken.

Enerzijds onderzoeken we of de combinatie van DSMs en het Net Option Value (NOV) model kan gebruikt worden als een kwantitatieve methodologie voor het onderzoeken van modulariteit in software ontwerpen. Om deze methodologie te evalueren introduceren we een software programma om ze te ondersteunen en passen we ze toe om een vergelijkend onderzoek te doen op aspect-georiënteerde en object-georiënteerde implementaties van design patronen. Op basis van deze experimenten formuleren we kritieken met betrekking tot de toepasbaarheid van het NOV model als een modulariteitsmetriek voor software.

Anderzijds onderzoeken we of DSM diagrammen kunnen dienen als een basis voor een nieuw soort software applicatie ter ondersteuning van software ontwikkelaars. We formuleren vereisten en een concrete aanpak voor een DSM-gebaseerde source-code browser voor object-georiënteerde software ontwikkeling. Om deze aanpak te valideren presenteren we een prototype van een dergelijke applicatie en demonstreren we hoe deze kan gebruikt worden in realistische situaties.

Abstract

In this dissertation we present our research into applications of Design Structure Matrices (DSMs) and related techniques in the context of software development. We pursue two research angles.

On the one hand, we investigate whether the combination of DSMs and the Net Option Value (NOV) model can be used as a quantitative methodology to assess modularity in software design. In order to evaluate this methodology we introduce a tool to facilitate its use and we apply it to conduct a comparative assessment on aspect-oriented and object-oriented design pattern implementations. Based on these experiments we formulate critiques with regard to the applicability of the NOV model as a modularity metric for software.

On the other hand, we investigate whether DSM diagrams can serve as a basis for a novel kind of support tool for software developers. We formulate requirements and a concrete approach for a DSM-based source-code browser for object-oriented software development. In order to validate this approach we present a prototype implementation of such a tool and we demonstrate how it can be used in real-world situations.

Acknowledgements

I would never have completed this dissertation without the support I received from a number of people. Therefore I would like to express my gratitude towards:

Prof. Dr. Theo D'Hondt, for promoting this thesis.

My advisors Dr. Andy Kellens and Dr. Johan Brichau, for coming up with the initial ideas for and contributions to the research that is covered by this dissertation. I would also like to thank them for proofreading and commenting on the text and for the dedication and patience they showed while I was writing it, as well as during my apprenticeship at the lab.

My girlfriend Sarah, for her invaluable support, her patience and her love. I will do everything in my power to make up for many long weeks she had to miss me while I was slaving away at this dissertation.

My parents, for their unconditional support and encouragement and for giving me the possibility to obtain a second higher education degree. I would also like to thank my mother for supplying me with provisions and my farther for his last-minute proofreading.

My brother Jasper, for his support and for the fun phone calls during his and my absence.

Andreas, Axel, Erwin, Jonas, Julie, Matthias, Pieter, Rien and my other friends in Ghent, for distracting me and for constantly enquiring when I would join them to go out again.

Karlien, for her companionship and support during our daily long working sessions at the lab in the course of the best part of July and August.

Clément, Dries, Erik, Isabel, Karlien, Lode, Pierre, Pieter, Thomas and my other fellow students and members of Infogroep, for making the past three years at the university such a fun time.

Everyone at the Programming Technology Lab, for providing a pleasant and serene working environment.

The Vrije Universiteit Brussel and its Departement of Computer Science, for providing an excellent education.

And last but not least, my cat Puck, for always cheering me up.

Contents

Chapter 1

Introduction

1.1 Context

In aiming for reduced complexity, higher quality, better reusability and lower development and maintenance costs, software development has evolved through a number of paradigms. Since the days of machine-level assembly languages, the procedural, functional and object-oriented paradigms – among others – have been introduced. Each of these paradigms brought new abstraction mechanisms, programming language features and design techniques.

The power of an abstraction mechanism is that it provides a means of modularisation, which allows software to be constructed from separate, cooperating modules. Modularity is perhaps the most important property developers strive for during the design and implementation of computer programs. The reason is that – in line with grand principles such as information hiding [55], the separation of concerns [18], encapsulation, isolation, etc. – modularity reduces overall complexity and increases reusability, evolability and maintainability.

Although the evolution of programming languages and design techniques has increased our ability to design software with higher modularity, there is still ample room for improvements. For instance, the vast size and complexity of many software development projects can quickly cause developers to lose track of the situation. Moreover, there are facets of computer programs that cannot be modularised using existing paradigms. Consequently, research into new paradigms continues to flourish. Recently the aspect-orientation paradigm, which claims improved modularisation properties over older paradigms by providing modularisation mechanisms for crosscutting concerns [36], has seen a great deal of attention.

While it is widely accepted that it is beneficial to design software systems in a modular fashion, it is not trivial to quantify the degree of modularity and other perceived qualities of software designs. By extension, claims with regard to the modularisation properties of new software development paradigms are difficult to validate. Therefore it generally takes a substantial amount of time for the advantages of new paradigms to become widely recognised.

However software developers need more than advanced languages and design techniques. They also rely heavily on a diverse set of support tools to make their lives easier. Examples include intelligent integrated development environments (IDEs), source-code browsers, documentation systems, versioning and collaboration systems, case tools, refactoring tools, testing frameworks, etc. Various kinds of diagrams, models and visualizations also play an important supporting role in every phase of the software development cycle and allow to bridge and integrate those phases.

The importance of support tools should not be underestimated and we believe that software development as a discipline can benefit just as much from innovations in this context as it can from research into new paradigms.

1.2 DSMs for Software Development

In this dissertation we investigate how Design Structure Matrices (DSMs) and related techniques can be applied to support software development in a broad sense. DSM diagrams were conceived [67, 68] to help manage the complexity of technical systems. They provide an abstract visualisation of the constituent parts of a system and the dependencies among those parts.

We believe that DSMs offer interesting opportunities for innovations in software development. We see possible applications on two fronts. On the one hand, we believe that DSMs offer an innovative way to evaluate modularity in software design. On the other hand, we believe that DSM diagrams can serve as a basis for a novel kind of support tool for software development. On both fronts, it is the focus on the modular structure of systems and the explicit depiction of – the distribution and nature of – dependencies, which make DSMs interesting in the context of software development.

1.2.1 Evaluating Modularity with DSMs and NOV

The structural elements of software need to cooperate to perform the functionality users expect. Such cooperation unavoidably introduces dependencies among those elements. Striving for modularity in software design means that structural elements are arranged in units called modules, where the goal is to maximise the dependency – or coupling – of elements within a module and to minimise the dependency to elements in other modules. Clearly there are degrees of dependency, thus there are degrees of modularity. Different designs for a software system can achieve different degrees of modularity. So if modularity is to be maximised, not all designs are equally good.

We believe that DSMs can offer the necessary insight to study and reason about the phenomena of modularity and dependency in software design. Moreover, the combination of DSMs with a related technique called Net Option Value (NOV) [5], could constitute a powerful metric to quantitatively evaluate the degree of modularity achieved by software designs.

1.2.2 DSM-based Support Tools

The issue of modularity and dependency should not be forgotten once the design for a software system has been decided upon. Software developers should always be aware of the prevailing dependencies in implementations and they should try improve modularity by minimising dependencies throughout the lifecycle of the product.

We believe that DSM-based support tools for software development offer new opportunities because DSMs communicate information on prevailing dependencies in a clear and concise way. Furthermore, we expect that DSM visualisations can facilitate the identification of opportunities for dependency minimisations.

1.3 Approach

In our work we approach the application of DSMs in software development from the two angles introduced above.

The first part of our research covers an evaluation of the combination of DSMs and NOV as a methodology for quantitative assessment and comparison of modularity in software designs. Inspired by the work of Lopes & Bajracharya [44, 45], we take the opportunity to link this exploration to the popular Aspect-Oriented Software Development (AOSD) research field. To evaluate the combination of DSMs and NOV, we apply this methodology to assess the claimed modularity advantages of the aspect-orientation paradigm over the object-orientation paradigm. We use aspect-oriented and object-oriented implementations [24] of the GoF design patterns [22] as the subject for this case study.

The second part of our research explores the use of DSM diagrams as a basis for support tools for Object-Oriented Software Development (OOSD). Starting out from observations concerning the discipline of OOSD, we develop a rationale for a DSM-based, IDE-integrated source code browser with metaprogramming facilities, to effectively support developers in common tasks related to dependency management. We provide a prototype implementation of such a tool to perform experiments and to validate our claims in real-world scenarios.

1.4 Outline of the Dissertation

This document is structured in accordance with the dual approach we follow in our work. Each research angle is covered by two chapters, which are preceded by an essential introductory chapter on DSMs.

Chapter 2 presents an in-depth introduction into DSMs, their origins and applications and related techniques such as NOV.

The next two chapters cover our research into to application of DSMs and NOV as a methodology for quantitative assessment of modularity in software designs. In Chapter 3 we introduce a novel software tool which is intended to facilitate experimentation with DSMs and the NOV model. In Chapter 4 we present an evaluation of the merits of the NOV model as a modularity metric for software, based on experiments on aspect-oriented and object-oriented design pattern implementations.

The following two chapters are about our research into the application of DSM diagrams as a basis for support tools for OOSD. In Chapter 5 we investigate the possibilities and requirements for such innovative tools and we introduce a prototype implementation. In Chapter 6 we present case studies which demonstrate how the prototype, or future similar tools, can assist software developers in common tasks.

We conclude this dissertation in Chapter 7, which provides a summary of our work and contributions, followed by an overview of future research directions and our final concluding remarks.

1.5 Notable Contributions

In this dissertation we present the following original contributions:

- A comprehensive introduction to DSMs and related techniques such as NOV, which prepares readers for passive and active use of DSM diagrams for both academic and professional purposes;

- A novel software tool which facilitates the application of the DSM+NOV methodology for design assessments, as well as exploration of the methodology itself. The tool introduces an innovative technique to assess system designs by focussing on two hierarchical levels;

- An evaluation of the DSM+NOV methodology as a technique for qualitative assessment of software. This work introduces novel ideas for analysis of software using the NOV model, but also leads to important questions regarding the applicability of the model as a modularity metric for software design;

- A rationale for a novel DSM-based support tool for OOSD and a prototype implementation of such a tool. This work introduces a unique approach which combines tree-based DSM visualisations and metaprogramming-based analysis facilities in an extendable, IDE-integrated source-code browser.

- Case studies which demonstrate how DSM-based source browsers can assist software developers in real-world tasks, most importantly in finding and evaluating modularisation opportunities by means of ad-hoc metaprograms.

Chapter 2

Design Structure Matrices

In this chapter we present a thorough introduction to Design Structure Matrixes (DSMs), which is essential in view of the following chapters. The main topics we cover are the origins and basic characteristics of DSMs and an in-depth discussion of the theory of Baldwin & Clark – which combines DSMs with a mathematical model, called Net Option Value (NOV), to quantitatively assess the economic value of designs.

2.1 Introduction

Due to the technological advances that dominate our society and economy, the complexity of the products, processes, organizations, and markets that surround us is ever-growing. Complexity is a major obstacle to design and develop successful products and services, but complex systems also facilitate better functionality and enable otherwise impossible innovations. Consequently, good complexity management can be a vital competitive advantage to any organization [91].

The Design Structure Matrix (DSM)[1] is a complexity management technique that has proved to be valuable for managing, designing, modelling, analysing and optimising technical systems, complex organisations, sizeable engineering projects, densely networked processes and large market structures [91].

A DSM is a diagram that visualises a system or a project using a compact matrix representation. The matrix confronts the constituent parts (parameters, subsystems, activities, tasks, ...) and indicates the dependencies among those. Depending on the context, the dependency patterns can represent different aspects of the system or project.

DSMs where first conceived by Donald Steward at General Electric in the late 1960s [67],

[1] Design Structure Matrices are also known as Dependency Structure Matrices, Problem Solving Matrices and Design Precedence Matrices.

but it took until 1981 before his work was published [68]. In his work Steward proposed a novel method, called the Design Structure System, to manage the complexity of large systems or engineering projects using DSMs.

Despite being conceived as a project management tool, a DSM is an analysis and design instrument that lends itself to a multitude of other applications across a wide range of domains and disciplines. The last decennium has seen modest, but growing, interest for DSMs from computer science, and software development in particular.

Although DSMs are well over 25 years old, they continue to attract the attention of academics and engineering professionals alike. The DSM community maintains a portal website [91] which lists all publications and centralises all knowledge on the subject. The community meets at the annual International DSM Conference [89], that has been sponsored by large corporations such as Boeing and the BMW Group.

In this chapter, we first explain the basic concepts of DSMs in section 2.2. Next, section 2.3 provides a detailed introduction to the work of Baldwin & Clark [5], who present a theory, supported by a DSM-based model, about the economic aspects of modularly in the design of complex systems. Finally, in section 2.4 we give background information on different roles and types of DSMs in applications in different fields. We conclude in section 2.5.

2.2 Basic Concepts

2.2.1 A Matrix of Parameters

A DSM of a particular system or project is a square adjacency matrix of (design) parameters, each of which represents a part of that system or project, and thus a source of variation in the overall system design or project planning. A system consisting of n such parameters results in a $n \times n$ Design Structure Matrix:

$$D = (d_{i,j})_{n\times n} = \begin{bmatrix} * & d_{1,2} & \dots & d_{1,n} \\ d_{2,1} & * & \dots & d_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n,1} & d_{n,2} & \dots & * \end{bmatrix}$$

2.2.2 Dependencies

The elements of the matrix represent the dependencies between the parameters. For example, a dependency from the x-th parameter to the y-th parameter (x-th parameter depends on the y-th) is represented by the value of the matrix element on the x-th row and the y-th column ($d_{x,y}$). DSMs do not account for dependencies between a parameter and itself, therefore the elements on the diagonal ($d_{i,j}$ with $i = j$) are usually masked with an "*".

Based on the dependencies among a pair of parameters the pair is said to be either parallel (no dependency between the parameters), sequential (one parameter depends on the other; there is a hierarchical relationship) or coupled (the parameters are mutually dependent; no hierarchical relationship), as shown in table 2.1.

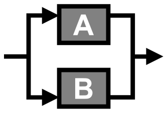
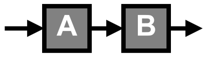
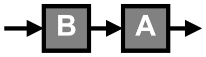
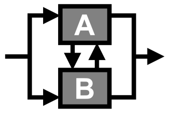
<table>
<tr><th></th><th>Parallel</th><th colspan="2">Sequential</th><th>Coupled</th></tr>
<tr><td>In words</td><td>A and B are independent</td><td>B depends on A</td><td>A depends on B</td><td>A and B are interdependent</td></tr>
<tr><td>Graph Representation</td><td> </td><td> </td><td> </td><td> </td></tr>
<tr><td>DSM Representation</td>
<td><table><tr><td></td><td>A</td><td>B</td></tr><tr><td>A</td><td>*</td><td></td></tr><tr><td>B</td><td></td><td>*</td></tr></table></td>
<td><table><tr><td></td><td>A</td><td>B</td></tr><tr><td>A</td><td>*</td><td></td></tr><tr><td>B</td><td>X</td><td>*</td></tr></table></td>
<td><table><tr><td></td><td>A</td><td>B</td></tr><tr><td>A</td><td>*</td><td>X</td></tr><tr><td>B</td><td></td><td>*</td></tr></table></td>
<td><table><tr><td></td><td>A</td><td>B</td></tr><tr><td>A</td><td>*</td><td>X</td></tr><tr><td>B</td><td>X</td><td>*</td></tr></table></td>
</tr>
</table>

Table 2.1: Configurations of parameters in DSMs (based on [81] & [90])

As a concrete example, figure 2.1a shows a DSM for a familiar system: a house. We see that some parameters, namely the staircases, the walls, the floors above ground level and the electrical wiring, depend on other parameters, while others, namely the foundations and the hook-up to the electric grid, are self-reliant. In a sense, the foundations constitute the most important parameter in the system, as they have the most dependant parameters.

		1	2	3	4	5	6
Foundations	1	*					
Staircases	2	1	*		1		
Walls	3	1	1	*		1	
Floors (above ground)	4	1		1	*		
Electricity grid hook-up	5					*	
Electric wiring	6			1		1	*

(a)

		1	2	3	4	5	6
Foundations	1	*					
Staircases	2	1	*		1		
Walls	3	1	2	*		2	
Floors (above ground)	4	1		4	*		
Electricity grid hook-up	5					*	
Electric wiring	6			1		2	*

(b)

Figure 2.1: A (binary) DSM (a) and an NDSM diagram (b) for the same "house"-system

The DSM in figure 2.1a and the ones in table 2.1 are binary DSMs, in which the matrix is populated with "1"s (or "X" marks) and "0"s (or empty cells). Because the dependency values are binary ($d_{i,j} \in \{0,1\}$), only the existence or absence of a dependency is taken into account, so a parameter is either dependant on or independent of another one. In other words, such DSMs do not account for *degrees of dependency*. By using numerical dependency values ($d_{i,j} \in \mathbb{N}^+$, or even $d_{i,j} \in \mathbb{R}^+$) instead, Numerical DSMs (NDSMs), can convey more information on the relationships between parameters than binary DSMs [81, 90]. Depending on the way they are used, numerical dependencies are called level numbers, importance ratings, dependency weights or dependency strengths. Figure 2.1b shows an example of an NDSM for our "house"-system, where, for instance, the dependency value of the floors above ground level to the walls is taken as 4, because each floor has four supporting walls.

2.3 Baldwin & Clark: DSMs & Modularity

In 2000, Carliss Baldwin and Kim Clark published a book titled *Design Rules, Volume 1: The Power of Modularity* [5], in which they develop a theory about the modularity in the design of complex systems. This novel theory approaches system design from an economic perspective.

The basic principle is that designing a system is a value-seeking process. When a system is designed in a modular fashion, implementations of individual modules can be designed in isolation – for instance by different (teams of) designers. Baldwin & Clark see this as a source of economic opportunities. Their theory introduces the innovative idea that *modularity* (see 2.3.1) in the design of a system creates or adds value in the form of *real options* (see 2.3.6.1). These options are considered sources of variation which enable one to improve the design, by experimenting with new implementations of individual modules. The possibility to substitute modules with alternative versions is described as a space of design options, each of which corresponds to a certain economic value that is modelled quantitatively.

The model relies on three components:

(a) the use of Design Structure Matrices as a visual representation of designs (see 2.3.2);

(b) a general theory of *modularity in design* (see 2.3.4), which introduces the concept of *design rules* (see 2.3.3) and six *modular operators* (see 2.3.5) as fundamental sources of design variation and evolution;

(c) *Net Option Value* (NOV) as a mathematical model to quantify the value of a modular design (see 2.3.6).

In their book [5], and in related publications [6, 7], Baldwin & Clark have demonstrated their theory by analyzing the influence of modularity on the evolution of computer hardware designs and the structure of the industry that creates them.

2.3.1 What is Modularity?

Baldwin & Clark define (in Chapter 3 of [5]) the concept of modularity using three subsidiary ideas. The first is the idea of *interdependence within and independence across modules*:

> *A module is a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units. Clearly there are degrees of connection, thus there are gradations of modularity.*

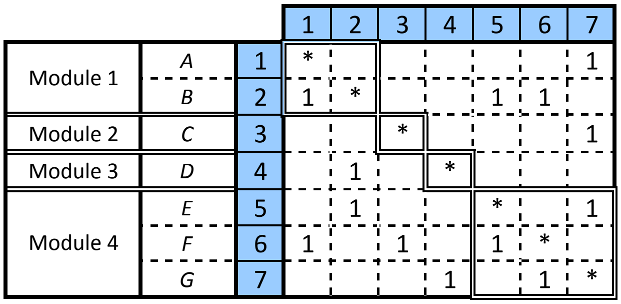
In other words, modules are units in a larger system that are structurally independent of one another, but work together to perform the functions of system. The system as a whole must therefore provide a framework – an architecture – that allows for both independence of structure and integration of function.

The second and the third idea capture the connection of modularity with three other, closely related, concepts: *abstraction*, *information hiding* and *interface*:

> A complex system can be managed by dividing it up into smaller pieces (modules) and looking at each one separately. When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate *abstraction* that has a simple *interface*. The *abstraction hides* the complexity of the element; the *interface* indicates how the element interacts with the larger system.
>
> *When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate "abstraction" with simple interface. The abstraction "hides" the complexity of the element. ...*

These general ideas align well with the notions of modularity, separation of concerns [18], abstraction, information hiding and interface that are common in software development. In fact, the principle of information hiding was first put forward in that context by David Parnas [55, 56], but it is general enough to be applied to any complex system.

2.3.2 Representing Modularity in Designs using DSMs



			1	2	3	4	5	6	7
Module 1	A	1	*						1
	B	2	1	*			1	1	
Module 2	C	3			*				1
Module 3	D	4		1		*			
Module 4	E	5		1			*		1
	F	6	1		1		1	*	
	G	7				1		1	*

Figure 2.2: A DSM with highlighted modules

Because they focus on the interdependencies among the parts of a system, DSM are well suited to study the modularity in designs.

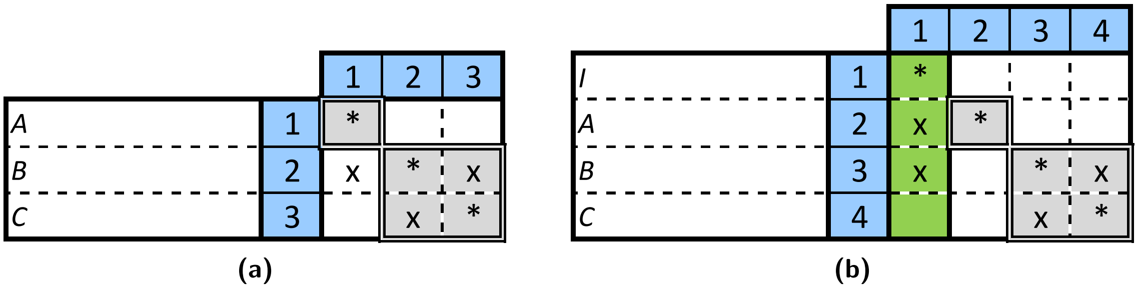
To highlight modules, formed by sets of design parameters (which correspond to their *structural elements*, see 2.3.1), their internal dependencies are usually surrounded by a thicker border, creating a series of boxes along the diagonal of the matrix. In the example shown in figure 2.2 an optional column to label the modules was added (sometimes such a column replaces the one with the parameter names). We see four modules in this DSM: Module 1 which contains parameters A and B, Modules 2 and 3 that consist of single parameters, respectively C and D, and Module 4 that groups parameters E, F and G.

We should note that Baldwin & Clark only use binary DSMs (no NDSMs) in their work.

2.3.3 Design Rules

In theory, modules should be structurally independent, but functionally integrated (see 2.3.1). Their structural independence allows them to be designed in isolation of one another, but the requirement of functional integration demands that the design of cooperating modules adheres to certain conventions (i.e. complies to a specified interface). Such conventions are the instrument by which information hiding is achieved, we called them *design rules* and they play a central role in the theory of Baldwin & Clark.

To illustrate the concept of design rules we should first introduce the notion of *proto-modular* and *modular designs*.



		1	2	3
A	1	*		
B	2	x	*	x
C	3		x	*

(a)

		1	2	3	4
I	1	*			
A	2	x	*		
B	3	x		*	x
C	4			x	*

(b)

Figure 2.3: DSMs for a proto-modular design (a) and a modular design (b)

When the parameters of a monolithic (non-modular) design are simply grouped into composites (called proto-modules) without a specific rearrangement of their dependencies, the resulting design is said to be proto-modular. Figure 2.3a shows an example (adapted from [72]) of a DSM for a proto-modular design. In theory, B-C, the group of parameters B and C, is not a *true* module because there is a structural dependency from parameter B to parameter A. To obtain a truly modular design, dependencies across proto-modules must be broken, through the use of a modular operator called *splitting*, one of six modular operators defined by Baldwin & Clark (see 2.3.5).

With splitting we introduce a new, separate, design parameter, called I (for interface), that will eliminate the direct dependency between A and B-C, permitting independent design choices for both modules. The DSM of the resulting modular design is shown in figure 2.3b[2]. We see that B no longer depends on A and instead both A and B take on a hierarchical dependency on the new parameter I. The latter dependency means that any value given to the B parameter (corresponding to a implementation for B) is constrained to access A only through the interface of A, represented by the value given to I. The former dependency signifies that the value for A can change freely without affecting B, as long as it complies to the interface of A, as specified by the value of I.

A chronological interpretation[3] of the ordering of the parameters in the DSM shown by figure 2.3b would demand that task I has been completed before tasks A, B and C are started. In other words, the designers first have to specify an interface for A, as a implementation of I, before modules A and B-C can be implemented. If the implementation of I is ever changed, the implementation of A and B will have to change as well, in order to comply to the new interface.

A design parameter, such as I, that decouples otherwise dependent proto-modules, is called a *design rule*. When a design rule is given a value, an assertion is made that that value is intended not to change. Therefore, design rules impose constraints that other parameters must respect and which they can assume to be stable. In the theory of Baldwin & Clark, modularisation[4] through the imposition of design rules is the key to constrain and structure the design space and search process [72].

The terminology of Baldwin & Clark makes a distinction between *visible* and *hidden* modules. Visible modules are modules that other modules depend on (they are "seen" by other modules). Design rules, such as I, fall in this category. Modules are called hidden when no other modules depend on them and they only depend on design rules.

Software engineers will likely notice that this theoretical approach to specifying interfaces, grasped by the concept of design rules, is consistent with common practices in (object-oriented) software design.

2.3.4 Modularity in Design

Now we have introduced design rules, we can define *modularity in design* as follows [7]:

> A complex engineering system is *modular-in-design* if (and only if) the *process of designing it can be split up and distributed across separate modules*, that are coordinated by *design rules*, not by ongoing consultations amongst the designers.

[2] The green box around the I column in figure 2.3b indicates that I is a design rule for this design.

[3] In 2.4.2 we will see that such an interpretation makes the diagram a *time-based* DSM.

[4] *Modularisation* can be defined as the evolution of a monolithical or proto-modular design to a modular design. In that sense, *a modularisation* specifies one way of doing that.

2.3.5 Modular Operators

The *splitting* operator we used in 2.3.3 is one of six modular operators that Baldwin & Clark introduce to model design evolution of complex systems. The others are: *substitution*, *augmentation*, *exclusion*, *inversion* and *porting*. The operators act as fundamental sources of variation in modular designs.

As we have seen, the *splitting* operator splits (proto-)modules and serves to modularise proto-modular designs. The other operators modify designs which are already modular. By applying the *substitution* operator the implementation of a module can be substituted for a new, adapted or improved one. In case of hidden modules substitution can take place without affecting other modules. The *augmentation* operator adds a module that was not part of the system before and its complement, the *exclusion* operator, removes a module from the system. The *inversion* operator standardises or collects common design elements across modules by organising the modules as a new hierarchical level. Finally, the *porting* operator transports a module of use in another system by creating "shell" around it [7, 72].

2.3.6 Net Option Value of a Modular Design

Even with design rules, a true modular design cannot always be achieved and designers often have a wide range of modularisations to choose from. Because not all modularisations are equally good (as there are gradations of modularity; see 2.3.1), it is useful to quantitatively evaluate designs. Therefore, Baldwin & Clark support their theory with a novel statistical model that reasons about the value added to a base system by modularity. This model, called Net Option Value (NOV), is based on the theory of real options and estimates the economic value of a modularly designed system by considering – in addition to other input parameters – the dependencies among the modules in the design, as documented by a DSM.

2.3.6.1 Real Options

In finance, an option is an investment which provides the holder with the *right* to make another investment in the future (by *exercising* the option), without an *obligation* to make that investment. Therefore, an option can have a positive payoff but never needs to have a negative one. Consequently, an option always has a positive present value, much like a lottery ticket [72].

Linking the economic theory of real options with the concept of modularity, Baldwin & Clark observed that modularity in design both multiplies and decentralizes real options that increase the value of a design. In a non-modular system, a design can only be replaced as a whole and the authority to accept changes is centralized. The designer only has one option: the choice to either accept or reject the whole design. On the other hand, in a modularised system, any or all modules can be redesigned and replaced independently. Here, replacement decisions are decentralised: the designers responsible for modules can make substitution decisions without coordination. In this sense, modularity provides a *portfolio of options* which, according to modern finance, always adds value [72].

2.3.6.2 NOV Formulae

We now provide a thorough explanation of the formulae of the NOV model. Table 2.2 shows a helpful overview and brief explanations of the symbols that appear in the equations below.

Symbol	Name/Explanation
NOV	(Total) Net Option Value of the system
S_0	Base value of the unmodularised system, usually normalised to 0
nov_i	Net Option Value of the i-th module
m	Number of modules in the system
σ_i	Technical potential of the i-th module
n_i	Complexity of the i-th module
k_i	The number of search/substitute experiments simulated for the i-th module
$C_i(n_i)$	The cost of one simulated search/substitute experiment on i-th module (a function of n_i), often taken as $C_i(n_i) = c_i n_i$
c_i	Redesign cost of the i-th module
Z_i	Visibility cost to replace the i-th module
$Q(k)$	Expected value of the best of k independent draws from a standard normal distribution, for all positive values in the distribution (see Appendix A)
$N(x)$	Standard normal cumulative distribution function (see Appendix A)
$n(x)$	Standard normal probability density function (see Appendix A)

Table 2.2: NOV formulae legend

The NOV model estimates the economic value of a modular design by looking at the added value generated by each module through the redesign option it represents. This is expressed by equation 2.1, which defines NOV, the (total) value of a design for a system.

$$NOV = S_0 + nov_1 + nov_2 + ... + nov_m \tag{2.1}$$

This formula should be interpreted as follows: splitting a design into m modules – each of which consists of a number ($\geqslant 1$) of design parameters – increases its base value, S_0, by the sum of the net option values (nov_i) of the resulting options. So for any i-th module, nov_i represents the value that the module contributes to NOV. These value contributions correspond to the opportunity each module i creates to invest in k_i experiments to create candidate replacements, each at a cost related to the complexity of the module, and, if any of the results are better than the existing choice, to *substitute* in the best of them, at a cost related to the visibility of the module to other modules [72].

Baldwin & Clark formalise the options value of each modular operator; for instance, how much is it worth to be able to substitute a module, augment it, exclude it, etc. So in fact, they define six formulas for nov_i, one for each modular operator. Additionally, they define a simpler, generic expression for nov_i, that values the option to redesign a module, no matter which modular operator is used. This expression is shown by equation 2.2.

$$nov_i = max\{\ max_{k_i}\{\underbrace{\sigma_i\sqrt{n_i}Q(k_i)}_{Benefit}\ \underbrace{-\ C_i(n_i)k_i\ -\ Z_i}_{Investment}\};\ 0\ \} \tag{2.2}$$

An overview of the formulae for all six operators is beyond the scope of this introduction to NOV, therefore we limit ourselves to this generic formula. However, for the sake of clarity, we explain it as if it only applies to the substitution operator[5]. In that sense, the formula describes the statistical simulation of search/substitute experiments. In the following paragraphs we present a breakdown of the formula, interpreted for the substitution operator.

Equation 2.2 takes nov_i, the net option value of the i-th module, as the *maximum* (*positive*) return value out of k_i design experiments on the i-th module. In other words, it is the expected payoff of exercising the search and substitute option for the i-th module *optimally*[6]. The maximisation is achieved by the max_{k_i} function. This function iteratively increments the number of (simulated) experiments k_i (e.g.: from 0 to 10) to find the so-called *break-even point*. This is the value for k_i that maximises the expected gain from the i-th module. The function then returns this maximal gain. Because usually only positive return values are taken into account [45], the first max function normalises negative nov_i values to 0.

The part of the formula within the braces of the max_{k_i} function corresponds to the expected return value from the i-th module for a particular value of k_i, accounting for both *benefit* and *investment* (option exercise costs).

The term $\sigma_i\sqrt{n_i}Q(k_i)$ represents the expected benefit to be gained by accepting the best candidate – a replacement for the existing module – generated by k_i independent experiments. The NOV model assumes this added value is a random variable normally distributed about the value of the existing i-th module choice (normalised to 0), with a variance $\sigma_i^2 n_i$ that reflects the *technical potential* σ_i and the *complexity* n_i of the i-th module. The standard deviation on the expected value is thus $\sqrt{\sigma_i^2 n_i} = \sigma_i\sqrt{n_i}$. The term $Q(k_i)$, formulised by equation 2.3, is the expected value of the best of k_i independent draws from a standard

[5] This is inspired by other authors who applied the NOV model in the context of software development [72, 73]; see Chapter 4 (section 4.3).

[6] By balancing the profits of design improvements with the costs of experimenting and redesigning (see economic principle of *diminishing returns*).

normal distribution, for all positive values in the distribution. Tabulated values of $Q(k_i)$ for $0 \leqslant k_i \leqslant 100$ can be found in Appendix A, along with additional information on this statistical distribution.

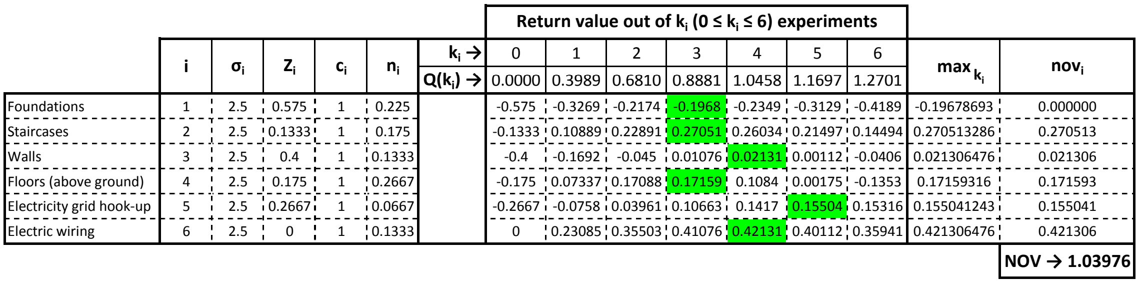
$$Q(k) = k \int_{0}^{\infty} x \; [N(x)]^{k-1} \; n(x) \; dx \tag{2.3}$$

$$Z_i = \sum_{j \text{ sees } i} c_j n_j \tag{2.4}$$

The investment consists of an *experimentation cost* and a *visibility cost*. The first element represents the cost incurred in doing the k_i experiments on the i-th module and is expressed by $C_i(n_i)k_i$. The cost of a single experiment on that module is expressed as a function C_i of n_i, the complexity of the module. This function is often, for instance in [45], taken as $C_i(n_i) = c_i n_i$, where c_i is the *redesign cost* of the i-th module. The second investment element for the i-th module is its visibility cost, is given by Z_i, which is computed by equation 2.4. This represents the cost to replace the i-th module, given the other modules in the system that directly depend on (or "see") it and the complexity n_j and redesign cost c_j of each of those. The Z_i parameter is the place where the NOV model takes the dependencies among the modules, as documented in a DSM, into account.

2.3.6.3 NOV Calculation Example

To clarify how the NOV formulae work, we return to the "house"-system example from section 2.2. The *NOV* of a design for a system can be calculated straightforwardly using a spreadsheet[7]. The one shown in figure 2.4 computes the *NOV* for the design of the house, as visualised by the (binary) DSM in figure 2.1a, where each individual parameter is interpreted as a module.



							Return value out of k_i ($0 \le k_i \le 6$) experiments								
	i	σ_i	Z_i	c_i	n_i	k_i →	0	1	2	3	4	5	6	max $_{k_i}$	nov_i
						$Q(k_i)$ →	0.0000	0.3989	0.6810	0.8881	1.0458	1.1697	1.2701		
Foundations	1	2.5	0.575	1	0.225		-0.575	-0.3269	-0.2174	-0.1968	-0.2349	-0.3129	-0.4189	-0.19678693	0.000000
Staircases	2	2.5	0.1333	1	0.175		-0.1333	0.10889	0.22891	0.27051	0.26034	0.21497	0.14494	0.270513286	0.270513
Walls	3	2.5	0.4	1	0.1333		-0.4	-0.1692	-0.045	0.01076	0.02131	0.00112	-0.0406	0.021306476	0.021306
Floors (above ground)	4	2.5	0.175	1	0.2667		-0.175	0.07337	0.17088	0.17159	0.1084	0.00175	-0.1353	0.17159316	0.171593
Electricity grid hook-up	5	2.5	0.2667	1	0.0667		-0.2667	-0.0758	0.03961	0.10663	0.1417	0.15504	0.15316	0.155041243	0.155041
Electric wiring	6	2.5	0	1	0.1333		0	0.23085	0.35503	0.41076	0.42131	0.40112	0.35941	0.421306476	0.421306
															NOV → 1.03976

Figure 2.4: NOV calculation for the DSM in figure 2.1a

The first column of the spreadsheet shows the names of the modules. The next five columns list the values, per i-th module, of the input parameters of the NOV calculation. In this example, the technical potential, σ_i, is assumed to be 2.5 for every module; the visibility

[7] Because setting up NOV spreadsheets manually is a tedious job, we created a tool to automate the process. We discuss this tool in Chapter 3.

cost, Z_i, is computed according to equation 2.4, by consulting the DSM in figure 2.1a to know which modules depend on (or "see") the module; the cost of an experiment on a module, $C_i(n_i)$, is taken as $c_i n_i$; the redesign cost, c_i, is set at 1 for every module and the complexities of the modules, n_i, are given values that sum to 1 and supposedly reflect the relative complexity of the parts of our imaginary house.

The middle part of the spreadsheet covers the search/substitute experiments. For every i-th module the expected return value (benefit minus investment) out of k_i experiments is computed for $0 \leqslant k_i \leqslant 6$. The benefits (given by $\sigma_i \sqrt{n_i} Q(k_i)$) increase with k_i. However, the experimentation cost (given by $c_i n_i k_i$) also increases with k_i and eventually the search/substitute experiments will meet diminishing returns. Consequently, after reaching the break-even point, the value for k_i that maximises the return value, higher values of k_i will decrease the return value. The spreadsheet highlights the return value at the break-even point of every module in green. The second-to-last column of the spreadsheet implements the max_{k_i} function; it selects the return value at the break-even point for each module. The last column copies those values but normalises negatives to 0, giving nov_i. Finally, all nov_i values are summed giving NOV, the total net option value for the design of the house.

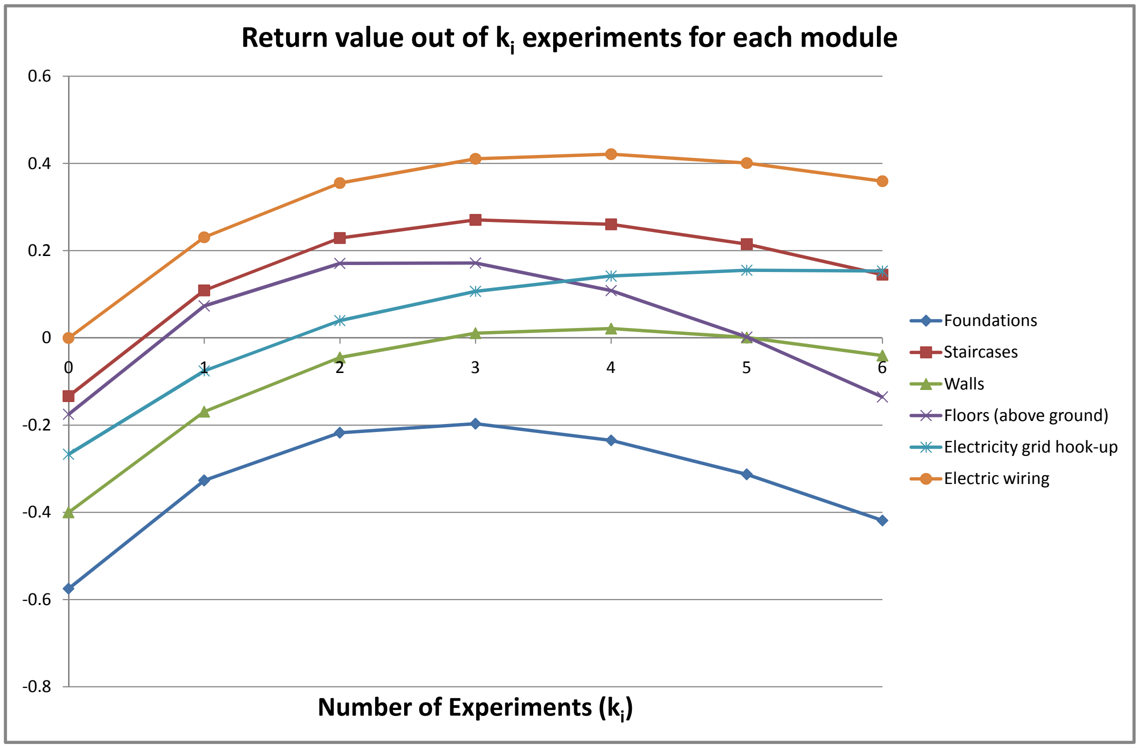


Figure 2.5: Graphed return values for k_i design experiments on each module from figure 2.4

As an illustration, figure 2.5 graphs the return value for every module from figure 2.4 in function of the number of search/substitute experiments. The peak in the curve for a specific module indicates the break-even point for that module.

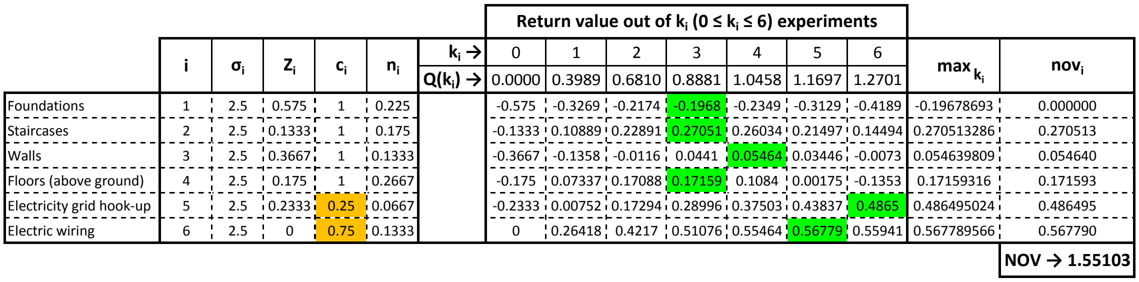
The graph and the spreadsheet clearly show that the sixth module, which represents the electric wiring, reaches the highest return value (on its break-even point at $k_6 = 4$) of all the modules. Hence, the module adds the most to NOV. The high return values of this module are caused by its lack of dependent modules (nothing depends on the electric wiring, hence the visibility cost Z_6 is 0) and its low relative complexity (given by n_6).

It also stands out that the first module, which represents the foundations, has the lowest return values. Because the investments outweigh the benefits (of finding better designs for the foundations), for any number of experiments (so for every value of k_1), the return values for the module are negative across the board. This is even the case at its break-even point (at $k_1 = 3$), which causes nov_1 it to be normalized to 0. Consequently, this module adds nothing to NOV. The low return values are due to the large number of dependents (three other modules depend on the foundations, resulting in a high visibility cost, given by Z_1) and the high relative complexity (given by n_1).

2.3.6.4 Strength and Weakness

The strength of NOV is that its mathematical expressions tie together modular dependencies, uncertainty and economic theory in a cohesive model [45].

However, the NOV model is not a black box. Its application in a particular field requires making assumptions, appropriate for the given context, about the input parameters used in the NOV calculation. Attributing different values to these parameters will obviously change the outcome of the NOV calculation, even when the design of the system, represented by a DSM, remains the same. As an illustration, figure 2.6 shows a new spreadsheet that again computes the net option value for the DSM in figure 2.1a, but uses different values, compared to the spreadsheet in figure 2.4, for the redesign cost of the fifth and the sixth module. The result is a 55% increase of NOV.



							Return value out of k_i ($0 \le k_i \le 6$) experiments								
	i	σ_i	Z_i	c_i	n_i	k_i →	0	1	2	3	4	5	6	max k_i	nov_i
						$Q(k_i)$ →	0.0000	0.3989	0.6810	0.8881	1.0458	1.1697	1.2701		
Foundations	1	2.5	0.575	1	0.225		-0.575	-0.3269	-0.2174	-0.1968	-0.2349	-0.3129	-0.4189	-0.19678693	0.000000
Staircases	2	2.5	0.1333	1	0.175		-0.1333	0.10889	0.22891	0.27051	0.26034	0.21497	0.14494	0.270513286	0.270513
Walls	3	2.5	0.3667	1	0.1333		-0.3667	-0.1358	-0.0116	0.0441	0.05464	0.03446	-0.0073	0.054639809	0.054640
Floors (above ground)	4	2.5	0.175	1	0.2667		-0.175	0.07337	0.17088	0.17159	0.1084	0.00175	-0.1353	0.17159316	0.171593
Electricity grid hook-up	5	2.5	0.2333	0.25	0.0667		-0.2333	0.00752	0.17294	0.28996	0.37503	0.43837	0.4865	0.486495024	0.486495
Electric wiring	6	2.5	0	0.75	0.1333		0	0.26418	0.4217	0.51076	0.55464	0.56779	0.55941	0.567789566	0.567790
															NOV → 1.55103

Figure 2.6: NOV calculation for the DSM in figure 2.1a, using adjusted values for c_5 and c_6

A weakness of the NOV model is that the meaning of its parameters in a given context can be unclear, which makes assumptions for their values hard to justify. For example, how does one measure the "technical potential" of a staircase? Clearly, the meaning of the parameters in a given context needs to be determined before the model can confidently applied. The same is true for NOV, the outcome of the computation. For instance, it is unlikely that an architect knows what a net option value of 1.03976 signifies for the house he is designing.

This turns out to be an important issue when applying DSMs and NOV in the context of software development, as we discuss in Chapter 4.

Another remark we should make is that the NOV model might not be suitable to be combined with NDSMs. Anyhow, we can assume that it was not designed for that purpose, since Baldwin & Clark only use it with regular, binary, DSMs throughout their publications (as noted in 2.3.2).

2.4 Overview of Applications

This section provides a broad outline of some of the applications of DSMs that have been developed in engineering at large and in other fields. This is not intended as a thorough survey, but rather as a brief introduction to ongoing research into applications of DSMs.

2.4.1 Roles of DSMs

According to [91], a DSM is both a project management tool and system analysis tool. In the role of a project management tool a DSM is primarily used to diagram information flows in complex projects. As a system analysis tool DSMs are used to analyse processes and architectures of products or organisations. However, there is no clear boundary between both roles, as specific applications of DSMs can combine them.

2.4.1.1 Project Management Tool

The Design Structure Matrix has its origins in project management [67, 68] and is still used in that context by large corporations such as General Motors, Boeing, Airbus and Intel. Project management applications of DSMs continue to receive interest from the research community as well [19, 81].

In these applications, DSMs represent the constituent stages, tasks or activities of an engineering project, along with the corresponding dependencies. The dependencies define the input which is required to start a certain activity and the generated output that needs to feed into other activities. Both input and output represent pieces of information. Hence, the pattern of dependencies in the DSM explicitly describes the exchange of information that is vital to the project. Such DSM-based project representations result, usually after being analysed and restructured with special algorithms (see 2.4.2), in an improved and more realistic execution schedule for the corresponding activities [91].

Traditional project management tools, such as PERT charts, Gantt charts and Critical Path methods (CPM)[8], were created to model and manage sequential and parallel processes consisting of discrete tasks that make up large construction projects. They capture work flow, often using pre and post conditions (e.g.: "Which tasks must be completed before task X can start?"), but do not to track the flow of information (e.g.: "Which pieces of information are needed before task X can succeed?").

According to the adepts of DSMs [19, 81], these tools are not suitable for managing large innovative projects because they fail to address the inherent complexity of such projects. For example, the design and development of complex "high-tech" products commonly requires a collaboration between a large number of participants from diverse backgrounds, resulting in complex relationships among both people and tasks. These relationships involve interdependencies (e.g.: task sequencing, feedback, cyclic dependencies, iterations, ...) which are hard to represent with tools lacking support for tracking information flow.

Compared to conventional project management tools, DSMs focus on representing information flows rather than work flows. Therefore, the DSM method, which is essentially an information exchange model, enables managers and product development planners to deal with the complex relationships in large engineering projects [19, 81].

2.4.1.2 System Analysis Tool

DSMs can also be applied as a tool to analyse complex systems. Analysed system architectures can be for both tangible (e.g.: material products) and intangible things (e.g.: projects or organisations). The compact and clear representation DSMs provide facilitates the capturing and understanding of interactions, interdependencies and interfaces between the elements of the system, such as subsystems or modules. Moreover, the diagrams can highlight key processes and enable engineers to discover previously unknown patterns in architectures. The diagrams can also show where staff members fit in the larger project or organisation they are part of [90].

2.4.2 Types of DSMs

Tyson Browning [14, 15, 90] distinguishes four different types of DSM applications, based on the kind of data that is represented. He also introduced two main categories if DSMs: static and time-based. Figure 2.7 shows a taxonomy of the categories and types of DSMs according to Browning.

In static DSMs the parameters represent the elements of a system that exist simultaneously, such as components of a product architecture or groups of people in an organisation. In time-based DSMs the parameters represent activities or processes, and their ordering in the

[8] Refer to [51] for an in-depth introduction to these and other conventional project management tools.

matrix indicates a flow through time, or in other words, the chronological order in which they are to be carried out.

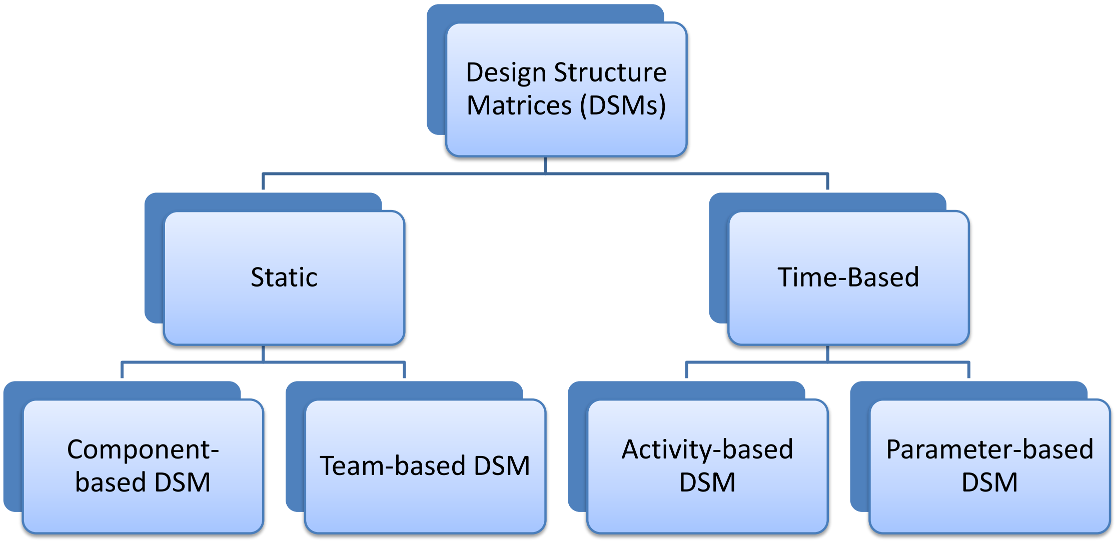


Figure 2.7: A taxonomy of DSM types according to Browning, adapted from [14, 15]

DSMs are often processed with metrics or algorithms that analyse and/or restructure the representation of a project or a system. In [15], Browning discusses each of the four types of DSMs and their accompanying analysis methods using industrial examples.

Component-based or Architecture DSMs are useful for modelling system architectures, involving relationships and interactions among components or subsystems, and for facilitating appropriate decomposition strategies. Component-based DSMs can be combined with clustering algorithms, which localise dependencies by defining subsets of parameters with minimal external dependencies [57]. Such subsets are called clusters or chunks, but largely correspond to the definition of modules (see 2.3.1).

Team-based, People-based or Organisation DSMs are used to design integrated organisation structures based on (groups of) people and their interactions. These DSMs can also be combined with clustering algorithms [47].

Activity-based, Task-based or Schedule DSMs are suited to model the information flow and other dependencies among processes and their constituent activities. (Re)Sequencing methods can be used to optimise the chronological order of the activities in activity-based DSMs [13, 82, 19]. A example of such an algorithm is partitioning, which transforms the matrix into a nearly lower triangular form in order to minimise feedbacks (activities that depend on activities in the future).

Finally, Parameter-based or Low-Level Schedule DSMs are aimed at modelling and integrating low-level design decisions and processes based on physical design parameter relationships. A noteworthy example can be found in the work of Black et al. who applied parameter-based DSMs to an automobile brake system design [9].

2.5 Conclusions

In this chapter we provided an extensive introduction to Design Structure Matrices (DSMs), the central subject of this dissertation.

The presentation of the theory of Baldwin & Clark, about the economic aspects of modularity in the design of complex systems, covered the best part of the chapter. We gave an overview of the principle elements of the theory: Baldwin & Clark's theoretical approach to modularity, the use of DSMs to visualize modular designs, the concept of design rules and most importantly, the Net Option Value (NOV) model. We explained and demonstrated how this mathematical model can be applied to a DSM representation of a design for a system, to quantitatively assess the economic value of that design.

We also paid attention to the many existing applications of DSMs, primarily in the field of engineering.

In the remainder of this dissertation we present our research into applications of DSMs and related techniques such as NOV. Chapter 3 introduces a tool we developed to facilitate experimentation with DSMs and the NOV model. In the following chapters we focus on applications of DSMs in the context of software development. Chapter 4 presents an assessment of the novel aspect-oriented software development (AOSD) paradigm, which was carried out using the tool from Chapter 3. Chapter 5 deals with our research into DSM-based support tools for object-oriented software development (OOSD) and introduces a prototype of such a tool. Finally, in Chapter 6 presents a number of real-world usage scenarios for that prototype.

Chapter 3

DSM+NOV Tool

In this chapter we introduce DSM+NOV Tool, a simple yet powerful application we developed to support experiments with DSMs and the NOV model. As far as we know, this is the first software of its kind. The main functionality of this tool is the automated generation of convenient spreadsheets that contain a DSM and handle all NOV calculations. We primarily created this application to generate the DSM+NOV spreadsheets we used to conduct a qualitative assessment of aspect-oriented and object-oriented design pattern implementations, which we discuss in the next chapter.

3.1 Introduction

In Chapter 2 we introduced the combination of Design Structure Matrices (DSMs) with the Net Option Value (NOV) model, as proposed by Baldwin & Clark [5]. This combination, which we will refer to as the *DSM+NOV methodology*, is promising for quantitative assessments of designs in diverse contexts. But the computations involved in the NOV model are too complex to be performed manually. However, as far as we know, there is no special-purpose software in existence today that handles NOV calculations[1], which clearly hampers experimentation with the DSM+NOV methodology.

This was problematic because we wanted to use the methodology to conduct a large-scale qualitative assessment of Aspect-Oriented and Object-Oriented Design Pattern implementations, which we discuss in Chapter 4. Therefore, we developed an application of our own, named *DSM+NOV Tool*, which we present here.

We first set out some requirements in section 3.2. Next, section 3.3 presents our approach, the automated generation of DSM+NOV spreadsheets. Then, section 3.4 gives an overview of the features and specific details of the application. Section 3.5 concludes the chapter.

[1] In Chapter 5 (section 5.5) we discuss the few DSM-related software tools we know of. However, none of these support NOV computation.

3.2 Requirements

The purpose of the application is twofold. On the one hand it should facilitate exploration of and experiments with the DSM+NOV methodology itself. On the other hand, in view of the work we present in Chapter 4, it should facilitate actual assessments of designs. Additionally, it should be a general-purpose tool, allowing the methodology to be applied in various fields.

3.3 Approach

Spreadsheets are an obvious way to automate NOV calculations. In Chapter 2 we showed an example spreadsheet (see figure 2.4) that implements the NOV formulae. We also notice that other authors [45, 46] follow this approach. If the DSM itself is added to such a spreadsheet the formulas for NOV calculation can be hooked up to it. That way, it is easy to experiment with different dependency patterns and adjusted values for NOV input parameters because the results are immediately visible.

However, even when templates are used, setting up such spreadsheets manually is a tedious job. Every time one needs to analyse a new system – with a different number of modules – or every time the number of simulated experiments must to be extended, all formulas need to be adjusted. This is both boring and prone to error.

To resolve this problem, we created an application, named *DSM+NOV Tool* [66], that generates ready to use customised DSM+NOV spreadsheets. The application was implemented as an add-in for Microsoft Excel [96] written in Visual Basic for Applications [104].

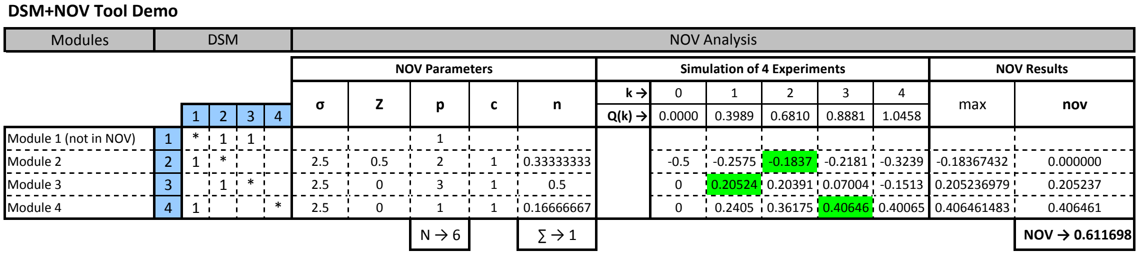
3.4 DSM+NOV Tool

DSM+NOV Tool generates spreadsheets, such as the one in figure 3.1 on the next page, that contain all necessary formulas and have a decent, ready-to-print layout. The spreadsheets consist of three parts: a listing that names the modules of the studied system, a DSM that documents the dependencies among those modules and a part that computes the NOV for the system.

The NOV[2] part lists or computes the values of the input parameters of the model, determines the outcomes of the simulated experiments (i.e.: the return value for each module for different numbers of experiments) and computes the final result. The return value at the break-even point for every module is highlighted in green. The NOV calculations are hooked

[2] Refer to Chapter 2 (2.3.6) for a full explanation of the Net Option Value model, its formulae and an extensive example.

up to the DSM, in which the dependencies need to be filled out manually, through the values of the Z parameter, the visibility cost. Further details on the computation of this parameter and on other choices with regard to NOV calculation are provided in 3.4.1.

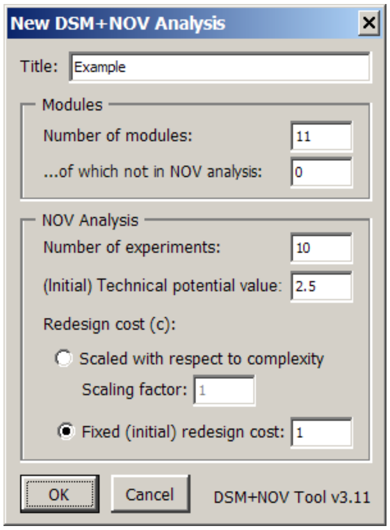


DSM+NOV Tool Demo

Modules	DSM					NOV Analysis												
						NOV Parameters					Simulation of 4 Experiments						NOV Results	
		1	2	3	4	σ	Z	p	c	n	k →	0	1	2	3	4	max	nov
											Q(k) →	0.0000	0.3989	0.6810	0.8881	1.0458		
Module 1 (not in NOV)	1	*	1	1				1										
Module 2	2	1	*			2.5	0.5	2	1	0.33333333		-0.5	-0.2575	-0.1837	-0.2181	-0.3239	-0.18367432	0.000000
Module 3	3		1	*		2.5	0	3	1	0.5		0	0.20524	0.20391	0.07004	-0.1513	0.205236979	0.205237
Module 4	4	1			*	2.5	0	1	1	0.16666667		0	0.2405	0.36175	0.40646	0.40065	0.406461483	0.406461
								N → 6		∑ → 1								NOV → 0.611698

Figure 3.1: Example of a generated spreadsheet

The tool generates new spreadsheets according to a configuration the user specifies by means of the dialog window shown in figure 3.2.



New DSM+NOV Analysis

Title: Example

Modules

Number of modules: 11

...of which not in NOV analysis: 0

NOV Analysis

Number of experiments: 10

(Initial) Technical potential value: 2.5

Redesign cost (c):

○ Scaled with respect to complexity

Scaling factor: 1

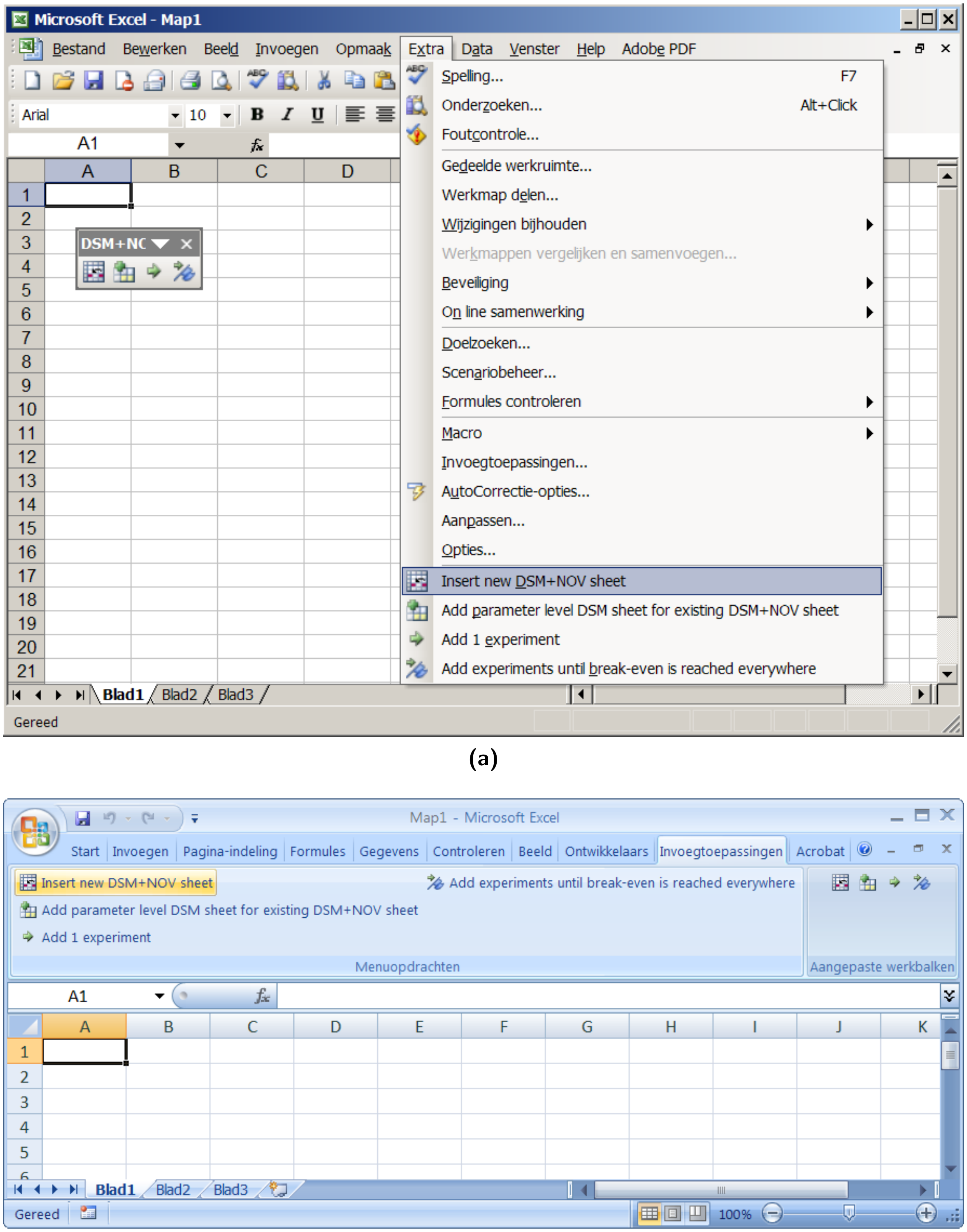
● Fixed (initial) redesign cost: 1

Figure 3.2: Dialog window to configure new DSM+NOV sheet

The DSM+NOV Tool add-in is intended for Excel 2003 and 2007 on Windows and Excel 2004 on Mac OS X[3] and fully integrates with the interface of the host application, as shown by the screenshots in figure 3.3 on the next page.

In addition to generating normal DSM+NOV spreadsheets our tool also contains functionality that allows systems to be documented on two hierarchical levels in parallel. We discuss this innovative feature in 3.4.2.

[3] Support for Excel on Mac OS X is untested; the add-in might also work in OpenOffice Calc (which supposedly has limited support for Excel add-ins).



(a)

(b)

Figure 3.3: Integration with interface of Excel 2003 (a) and 2007 (b) on Windows

3.4.1 Net Option Value Calculation

Here we list the most important choices for our implementation of the NOV model.

One of the options in the configuration dialog (see figure 3.2) allows users to exclude a number of modules from the NOV calculation. As shown in the example spreadsheet (see figure 3.1) NOV-exclusion is indicated in the name of such modules. The intention is to allow users to employ a DSM which includes dependencies from parts of the studied system to external entities which are not an integral part of the system, without affecting the NOV results. The inspiration for this feature comes from Sullivan et al. [72][4], who introduced *Environment and Design Structure Matrices* (EDSMs), a DSM variant extended with the notion of *environment parameters* (EPs).

The configuration dialog lets users specify the number of simulated experiments. However, because it can be hard to estimate beforehand the number of experiments that is needed to reach a break-even point for every module, our tool can also insert additional experiment columns into earlier generated spreadsheets. Users can choose to add experiments one by one or they can use the automatic break-even points finding feature, which adds experiments until the return value of all modules is decreasing (i.e.: until each module has reached its break-even point).

Currently our tool uses the generic formula to compute the net option value for each module. Future versions of the tool could offer the user a choice between the specific formulae for each of the six modular operators defined by Baldwin & Clark[5].

To compute the return value of k_i simulated experiments on each i-th module the value of $Q(k_i)$ needs to be known. However, computing that value involves integrations (see Appendix A), which Microsoft Excel cannot handle. Therefore our application uses a numerical integration algorithm. The code was taken from XNumbers [107], an open source library that extends Excel with support for numerical methods.

$$Z_i = \sum_{j \ sees \ i} c_j n_j = \sum_{j=1}^{i-1} d_{j,i} c_j n_j + \sum_{j=i+1}^{m} d_{j,i} c_j n_j \tag{3.1}$$

The visibility cost parameter, Z, links NOV calculation to the dependencies in the DSM. For the i-th module, it is computed according to equation 3.1. Where m is the number of modules in the system, c_j and n_j are respectively the redesign cost and the complexity of a j-th module and $d_{j,i}$ is the value on the j-th row and the i-th column in the DSM, corresponding to a dependency from the j-th module to the i-th module. Dependencies from excluded modules are discarded, as c_j and n_j equal to 0 for those modules.

[4] We discuss the work of Sullivan et al. in Chapter 4 (section 4.3).

[5] Refer to Chapter 2 (2.3.5) for an explanation on substitution and the other five modular operators.

Users can freely adjust the values of the remaining NOV input parameters for each module. But DSM+NOV Tool provides default settings which follow conventions and assumptions proposed by Baldwin & Clark or by authors that have applied the NOV model in the context of software development[6]:

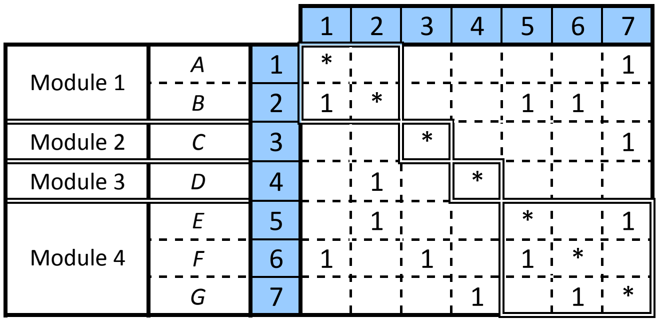
- The technical potential of each i-th module, σ_i, defaults to 2.5;
- The redesign cost of each i-th module, c_i, is either fixed at 1 or taken as a user specified factor (which defaults to 1) that is scaled with respect to n_i, the complexity of that module;
- The complexity of each i-th module, n_i, is taken as the number of design parameters contained in the module, p_i, divided by N, the total number of design parameters in the system (ignoring those in NOV-excluded modules);
- The number of design parameters in each i-th module, p_i, defaults to 1.

3.4.2 Module- and Parameter-level DSMs

The generation of parameter-level DSMs is the most innovative feature of DSM+NOV Tool. The goal is to allow dependencies occurring on a lower hierarchical level of the studied system to be documented and taken into consideration by the NOV calculation. To explain this feature we first need to recapitulate the difference between design parameters and modules.

Design Parameters and Modules

In Chapter 2 we explained that a DSM confronts design parameters to document the dependencies among them. Design parameters correspond to discrete structural elements in the design of the studied system. A module groups a set of design parameters, which are (expected to be) strongly dependant on each other but relatively weakly dependant on design parameters outside the module. Figure 3.4 reproduces the example we have used to show how DSM diagrams can be extended to visualise modules.



			1	2	3	4	5	6	7
Module 1	*A*	1	*						1
	B	2	1	*			1	1	
Module 2	*C*	3			*				1
Module 3	*D*	4		1		*			
Module 4	*E*	5		1			*		1
	F	6	1		1		1	*	
	G	7				1		1	*

Figure 3.4: A DSM with modules consisting of design parameters

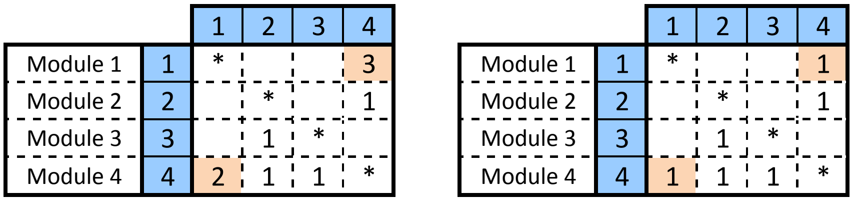
[6] Refer to Chapter 4 (section 4.3).

Conceptually both modules and (design) parameters represent parts of the studied system, but on different levels or layers of hierarchy, detail, abstraction or granularity. For example, when studying an object-oriented software system, modules could represent classes, while parameters could represent methods.

Module-level DSMs

The NOV model evaluates the design of a system based on properties of the modules in that system and the dependencies among those. In other words, NOV only looks at dependencies on the module level. Therefore, the DSMs in our generated spreadsheets differ from regular ones because they confront modules instead of design parameters and thus only document dependencies among modules.

In a sense, our *module-level DSMs* offer a "zoomed out" view on the system, compared to regular *parameter-level DSMs*, because they summarise dependencies among parameters as dependencies among modules. As an illustration, figure 3.5 shows two module-level versions for the system that is represented by the parameter-level DSM in figure 3.4.



		1	2	3	4
Module 1	1	*			3
Module 2	2		*		1
Module 3	3		1	*	
Module 4	4	2	1	1	*

(a)

		1	2	3	4
Module 1	1	*			1
Module 2	2		*		1
Module 3	3		1	*	
Module 4	4	1	1	1	*

(b)

Figure 3.5: Module-level versions of the parameter-level DSM in figure 3.4, created by summing (a) or by maximising (b) parameter-level dependencies

The difference between both module-level DSMs lies in the way parameter-level dependencies are summarised. In figure 3.5a the dependency from a module to another module is taken as the sum of all dependencies from parameters of the first module to those of the second module. In figure 3.5b the dependency from a module to another module is taken as the highest dependency from a parameter of the first module to one of the second module. The different summarising method causes two differing dependency values (as highlighted in the DSMs).

We should note that the DSM in figure 3.5a is an NDSM because summing the parameter-level dependencies results in two numerical dependency values (> 1). The DSM in figure 3.5b is still a regular one because maximising the (binary) parameter-level dependencies results only in binary dependency values.

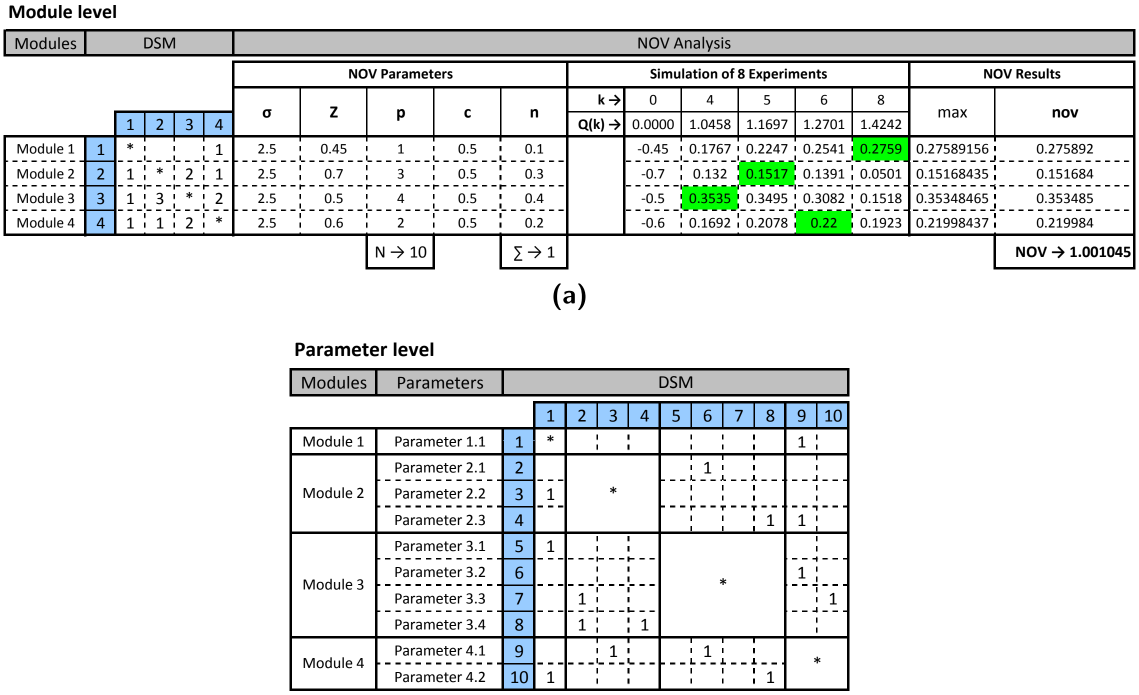
Combining Module- and Parameter-level DSMs

When using our application, we must fill out the DSM with dependencies on the level of modules, so if the studied system has a meaningful lower hierarchical level, we must either ignore it or manually summarise relevant dependencies occurring on that level. In either case, we lose a degree of detail because the parameter-level dependencies are not explicitly documented.

To resolve this problem, DSM+NOV Tool includes a feature that enables users to simultaneously document dependencies occurring on two hierarchical levels. This novel approach calculates the NOV of a system based on the combination of a module-level DSM and a parameter-level DSM.

The procedure is as follows. Starting off with a regular generated DSM+NOV spreadsheet the user must specify the number of design parameters contained in each i-th module, by adjusting the value of p_i. Next, the user can let the tool generate a parameter-level DSM – as a second spreadsheet – with the right number of parameters per module. In there, the parameter-level dependencies can be filled out, as illustrated by figure 3.6b.

To take the parameter-level dependencies into account for NOV calculation, the tools links both DSMs by inserting summarizing formulas into the dependency cells of the original,



Module level

Modules		DSM				NOV Analysis												
						NOV Parameters					Simulation of 8 Experiments						NOV Results	
						σ	Z	p	c	n	k →	0	4	5	6	8	max	nov
		1	2	3	4						Q(k) →	0.0000	1.0458	1.1697	1.2701	1.4242		
Module 1	1	*			1	2.5	0.45	1	0.5	0.1		-0.45	0.1767	0.2247	0.2541	0.2759	0.27589156	0.275892
Module 2	2	1	*	2	1	2.5	0.7	3	0.5	0.3		-0.7	0.132	0.1517	0.1391	0.0501	0.15168435	0.151684
Module 3	3	1	3	*	2	2.5	0.5	4	0.5	0.4		-0.5	0.3535	0.3495	0.3082	0.1518	0.35348465	0.353485
Module 4	4	1	1	2	*	2.5	0.6	2	0.5	0.2		-0.6	0.1692	0.2078	0.22	0.1923	0.21998437	0.219984
								N → 10		∑ → 1								**NOV → 1.001045**

(a)

Parameter level

Modules	Parameters		DSM									
			1	2	3	4	5	6	7	8	9	10
Module 1	Parameter 1.1	1	*								1	
Module 2	Parameter 2.1	2		*				1				
	Parameter 2.2	3	1									
	Parameter 2.3	4								1	1	
Module 3	Parameter 3.1	5	1				*					
	Parameter 3.2	6									1	
	Parameter 3.3	7		1								1
	Parameter 3.4	8		1		1						
Module 4	Parameter 4.1	9			1			1			*	
	Parameter 4.2	10	1							1		

(b)

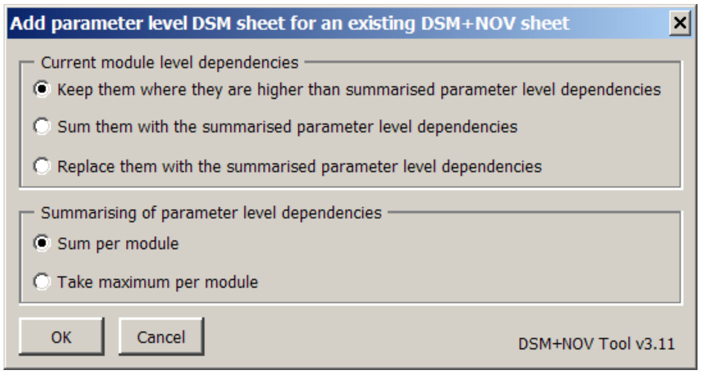
Figure 3.6: NOV calculation for a module-level DSM (a) which summarises dependencies in a parameter-level DSM (b) by summing

module-level, DSM – as shown by figure 3.6a. In the example in figure 3.6 the DSMs are linked through summing, but our tool supports summarising by maximising as well. Because the NOV model was not designed for NDSMs, as noted in Chapter 2, summarising by maximising is can be useful because it avoids numeric dependency values in the module-level DSM when the parameter-level DSM is binary.

Strict Module-level Dependencies

Some kinds of dependencies occur strictly on the level of modules, meaning they cannot be attributed to a pair of parameters. For instance, returning to our example from object-oriented software, it is obvious that inheritance is a relation (and thus a dependency) between classes, not between a pair of their methods. Therefore, we included a way to combine, instead of replace, module-level dependencies with summarised parameter-level dependencies.

To account for strict module-level dependencies, we must fill them out in the DSM of spreadsheet we start out with (the *temporary* module-level DSM), before the parameter-level DSM is generated. Then, upon the generation of the parameter-level DSM, the dialog window shown by figure 3.7 offers three combine/replace options, in addition to the choice between summarising by summing or by maximising.



Add parameter level DSM sheet for an existing DSM+NOV sheet

Current module level dependencies

● Keep them where they are higher than summarised parameter level dependencies

○ Sum them with the summarised parameter level dependencies

○ Replace them with the summarised parameter level dependencies

Summarising of parameter level dependencies

● Sum per module

○ Take maximum per module

Figure 3.7: Dialog window with options for the generation of a parameter-level DSM

The first option keeps module-level dependency values which are higher than their corresponding summarised parameter-level dependency value, and replaces them otherwise; the second option sums module-level dependency values with their corresponding summarised parameter-level dependency value; and the third option replaces all module-level dependency values with their corresponding summarised parameter-level dependency value.

Equation 3.2 and equation 3.3 show how the second option works, respectively in case of summarising by summing and summarising by maximising.

$$m_{i,j} = \mathcal{M}_{i,j} \ + \sum_{\forall x \in i,\ \forall y \in j} p_{x,y} \tag{3.2}$$

$$m_{i,j} = \mathcal{M}_{i,j} + \max_{\forall x \in i,\ \forall y \in j} p_{x,y} \tag{3.3}$$

In the equations:

- $m_{i,j}$, $\mathcal{M}_{i,j}$ and $p_{i,j}$ are respectively dependency values in the final module-level DSM, the temporary module-level DSM and the parameter-level DSM;
- i and j are modules, with $i \neq j$;
- x and y are design parameters, respectively contained in module i and module j.

3.5 Conclusions

DSM+NOV Tool significantly reduces the time one needs spend to prepare a DSM+NOV spreadsheet and thus clearly satisfies the requirements set out in section 3.2.

We believe the tool is a valid technical contribution to the field because it facilitates the application of the DSM+NOV methodology to concrete design assessments as well as exploration of and experimentation with the methodology itself. This is illustrated by the work we present in Chapter 4, which deals with comparative assessments, carried out using DSM+NOV Tool, of object-oriented and aspect-oriented software designs. The tool has also been shared with other researchers[7].

As far as we know, the idea to base NOV calculation on a combination of two DSMs, which document dependencies on different hierarchical levels, is a novel concept in the field. In Chapter 4 we demonstrate how we used this innovative feature to document implementation dependencies in software on two *granularity* levels.

[7] Namely Cristina Lopes and Sushil Bajracharya, whose work we discuss in Chapter 4 (section 4.3).

Chapter 4

Assessing Aspect-Orientation using DSMs and NOV

In this chapter we present an evaluation of DSMs and the NOV model as a methodology for quantitative assessment of modularity in software designs. To experiment with this methodology, we apply it to aspect-oriented and object-oriented implementations of the GoF design patterns, in order to assess the claimed modularisation advantages of the novel aspect-orientation paradigm. Based on those experiments we formulate critiques on the applicability of the NOV model as a modularity metric for software design.

4.1 Introduction

In Chapter 2 we introduced Design Structure Matrices (DSMs) and the Net Option Value (NOV) model [5]. We believe that the combination of both techniques is a promising methodology for quantitative assessments of designs in diverse contexts. In this chapter, we present our research into the application of this *DSM+NOV methodology* as a tool for quantitative assessment and comparison of software designs. Specifically, we aim to evaluate whether or not the NOV model can serve as a metric to quantify modularity in software designs. By extension, we want to investigate whether NOV can help to validate or invalidate the claims of software development paradigms with regard to modularisation properties.

Inspired by the work of others [44, 45, 73], we took the opportunity to link this research to the Aspect-Oriented Software Development (AOSD) field. The novel aspect-orientation paradigm claims to provide improved modularisation properties over the object-orientation paradigm. This makes aspect-orientation an interesting subject for our evaluation of NOV as a modularity metric.

In summary, we want to evaluate the merits of NOV as a modularity metric, by employing the DSM+NOV methodology to carry out a quantitative assessment of the modularisation properties of the aspect-orientation paradigm, in comparison with the object-orientation paradigm. As a concrete case study, we choose to compare aspect-oriented and object-oriented implementations of the GoF design patterns [22], respectively written in AspectJ and Java [24].

We present experiments on AspectJ and Java versions of three design patterns. The experiments were carried out using the DSM+NOV spreadsheet generation tool we introduced in Chapter 3. Based on the results of these experiments we formulate conclusions and critiques on the applicability of NOV as a modularity metric for software.

In this chapter, we first provide an elaborate introduction to the aspect-orientation paradigm in section 4.2. Next, in section 4.3 we discuss the work of others who have applied DSMs and NOV in the context of software development. Further, in section 4.4 we provide a brief introduction to design patterns. Then, section 4.5 deals with the experiments we conducted and formulates observations based on the results. We conclude this chapter in section 4.6.

4.2 Aspect-Oriented Software Development

In its quest for higher software quality and lower development and maintenance costs, software development is constantly evolving. In its history of over 60 years, the discipline has seen many different paradigms with accompanying programming languages and design techniques. Since the days of machine-level assembly languages, the procedural, functional and object-oriented programming paradigms have been introduced (among others). Each new paradigm provides new abstraction mechanisms and language features over older paradigms, allowing improved design and implementation structures, which ideally lower overall complexity and increase reusability.

One of the latest developments in the evolution of programming is the rise of Aspect-Oriented Programming (AOP), a novel paradigm which was introduced by Gregor Kiczales in the second half of the 1990s [32, 36, 37]. Aspect-orientation (AO) builds further upon the concepts of object-orientation (OO) – and older paradigms – but focuses on providing mechanisms to enable the modularisation of so-called *crosscutting-concerns* (see 4.2.1 and 4.2.2) by means of *aspectual decomposition* (see 4.2.3).

Aspect-Oriented Software Development (AOSD) is the broader paradigm devoted to applying the concepts of aspect-orientation to the whole software development lifecycle.

In the remainder of this section we will introduce the most important concepts of aspect-orientation. Further introductory reading on AOSD can be found in [12], upon which this section is based. An elaborate account on the history of AOSD is provided in [43].

4.2.1 Modularisation[1] and Separation of Concerns

The power of an abstraction mechanism is that it provides a means of modularisation, which allows software to be structured in separate, cooperating, modules. Modularisation helps software engineers achieve what is called the *separation of concerns* (SoC).

This principle, originally introduced by Dijkstra [18], is one of the driving ideas of the evolution of software development paradigms. In the context of software development, a concern is defined as an interest which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one of the stakeholders [75]. A concern of a software application can be related to both functional (e.g.: as captured by use cases) and non-functional (e.g.: reliability, scalability, ...) requirements.

The principle of separation of concerns demands that every concern is treated in isolation throughout the development lifecycle. This means that each concern should be modelled, designed and implemented as a separate unit or module. Thereby the complexity of individual modules is reduced and it is ensured that each module represents a well-defined subpart of the system, corresponding to a single concern. This results in improved evolvability and reusability of the application and its parts. Such properties help to keep software projects manageable, which can ultimately lead to lower development and maintenance costs.

In object-orientation modularisation is achieved by decomposing software applications into individual units called objects. Ideally any single concern is represented by a particular group of such objects. There are, however, concerns which cannot be modularised using the techniques the OO paradigm provides. Enabling the modularisation of such *crosscutting concerns* is the main goal of the AO paradigm.

4.2.2 Crosscutting Concerns

Common examples of crosscutting concerns include synchronisation policies, error and exception handling, enforcement of real-time constraints, security requirements, logging, tracing and fault tolerance mechanisms. Without a paradigm that explicitly handles such crosscutting concerns, their implementation causes *scattering* and *tangling* with respect to the implementation of other concerns.

Scattering is defined as the occurrence of the representation of one concern in multiple modules [75]. Tangling, on the other hand, is defined as the occurrence of representations of multiple concerns mixed together in a single module [75]. Both issues tend to appear together; they describe different facets of the same problem.

To illustrate scattering and tangling, we will show how they manifest themselves in the

[1] Refer to Chapter 2 (2.3.1) for general definitions of *modularity*, *modules* and *modularisation*.

implementation of logging, which has become the obligatory example crosscutting concern in texts on AOSD. The running example we will use, here and throughout this section, is set in a hypothetical[2] client-server banking application. Figure 4.1 shows a part of this system, implemented in an object-oriented programming language.

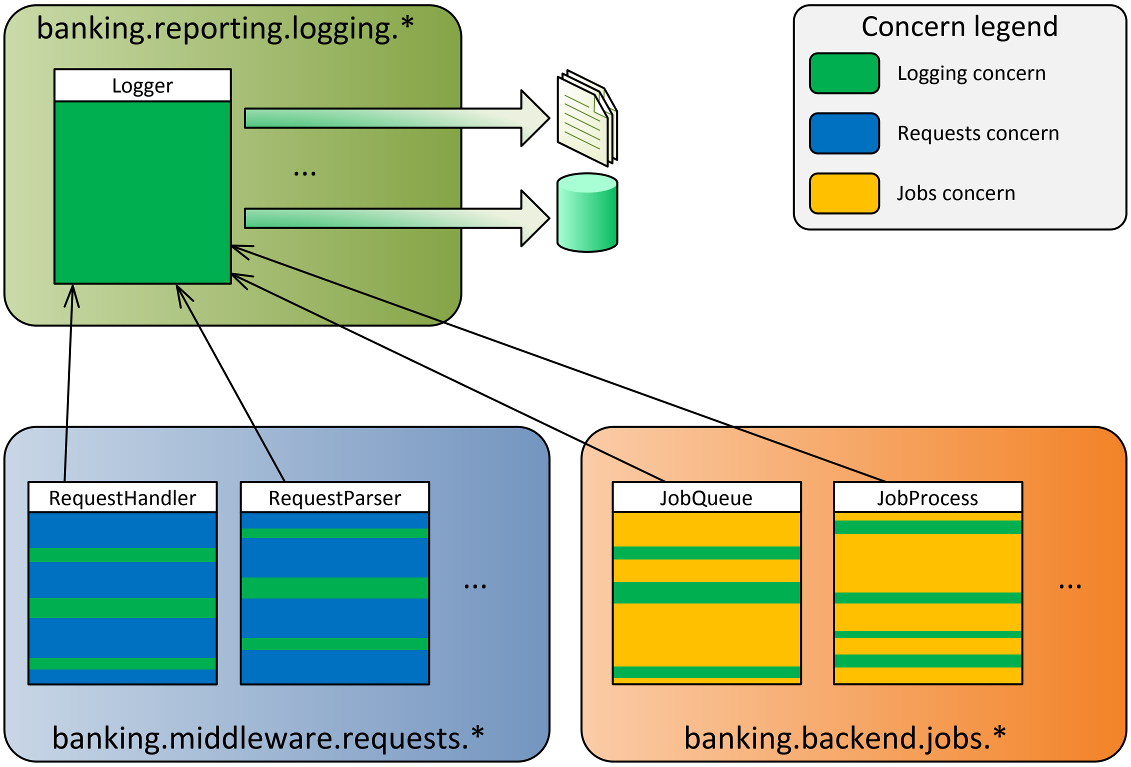


Figure 4.1: Crosscutting logging concern causes scattering and tangling

We can look at this system from two perspectives. From a structural perspective, the figure shows the arrangement of objects in three different groups[3]. These groups are modules that are assumed to be responsible for a specific functionality. One group, part of the reporting infrastructure of the system, holds objects that deal with logging. A second group, located in the middleware layer of the system, is responsible for handling requests from clients. The last group, situated in the backend of the system, manages batch processing jobs.

In parallel with the structural perspective, the system can be observed from the perspective of concerns. The figure shows, by means of colour coding, the actual code level locations, within each object, of the implementation of three concerns: logging, requests and jobs.

[2] An often cited real world case – which backs hypothetical examples such as ours – of the crosscutting nature of logging can be found in the Apache Tomcat web server [33].

[3] The groups of objects in our example are named in the style of Java packages. Packages are a mechanism for grouping of related classes.

It is clear that the implementation of the requests concern and the jobs concern is limited to the respective groups of objects. However, the implementation of the logging concern is not limited to objects in the logging group, on the contrary, there are bits of logging code in the objects that deal with the requests and jobs concerns as well. In terms of the definitions, the logging concern crosscuts the banking application as its representation is scattered ("spread out") over multiple modules and tangled ("mixed") with the representation of other concerns within individual modules.

We can conclude that object-oriented modularisation achieves the separation of the requests and jobs concerns, but fails to do so for the logging concern. Because the banking software is required to log diverse operations occurring in different parts of the system, including those that deal with requests and jobs, any object-oriented implementation will always, regardless of the chosen decomposition into objects, introduce logging code – which includes at least a method call to a logging object – at the location of every operation that must be logged. Hence, the implementation of logging will be distributed across multiple modules, causing scattering and tangling and making it a crosscutting concern.

Crosscutting concerns like logging are common in most software systems and developers might see them as being harmless. However, crosscutting concerns can be detrimental because they break the principle of separation of concerns: single modules can contain the implementation of multiple concerns (tangling) and crosscutting concerns lack a separate, explicit, representation (scattering). These issues hinder reuse and evolvability of both crosscutting and non-crosscutting concerns, which can be particularly harmful, especially in large-scale projects. The aspect-orientation paradigm provides a remedy for this problem by introducing new modularisation mechanisms that enable developers to explicitly represent crosscutting concerns as separate entities.

Although we have discussed scattering and tangling in an object-oriented application, it is important to note that the problem of crosscutting concerns is not limited to object-orientation, it can be observed in other paradigms as well. The truth is there are always restrictions on the ability to modularly represent particular concerns. These restrictions are intrinsic to the decomposition technique dictated by the paradigm [74].

4.2.3 Aspectual Decomposition & Aspects

Pre-AO paradigms have always relied on *functional (de)composition*. In [37], Gregor Kiczales pointed out that the resulting modules (e.g.: subroutines, procedures, functions, ADTs, components, objects) can be seen as *generalised procedures*. In order to modularise crosscutting concerns aspect-orientation introduces a fundamentally new modularisation technique, which is based on *aspectual (de)composition*. The resulting modules, representing crosscutting concerns, are called *aspects* and are no longer generalized procedures.

The difference lies in the invocation mechanism. The behaviour of (generalised) procedures is always explicitly invoked by other procedures. In contrast, aspects have an implicit invocation mechanism. This means their behaviour is implicitly invoked at specific points in the implementation of other modules. Strictly speaking, the programmers of other modules do not need to be aware of the crosscutting concern and its implementation as an aspect[4].

To achieve implicit invocation, aspects needs to specify themselves *where* or *when* their behaviour needs to be invoked. An aspect does this by autonomously observing the execution of the (base) program: when specific patterns of actions occur it can execute additional behaviour on its own. Following the common terminology [75], the actions of the base program are called *join points*, the patterns that describe sets of such join points are known as *pointcut(designator)s* and the additional behaviour is referred to as *advice (code)*.

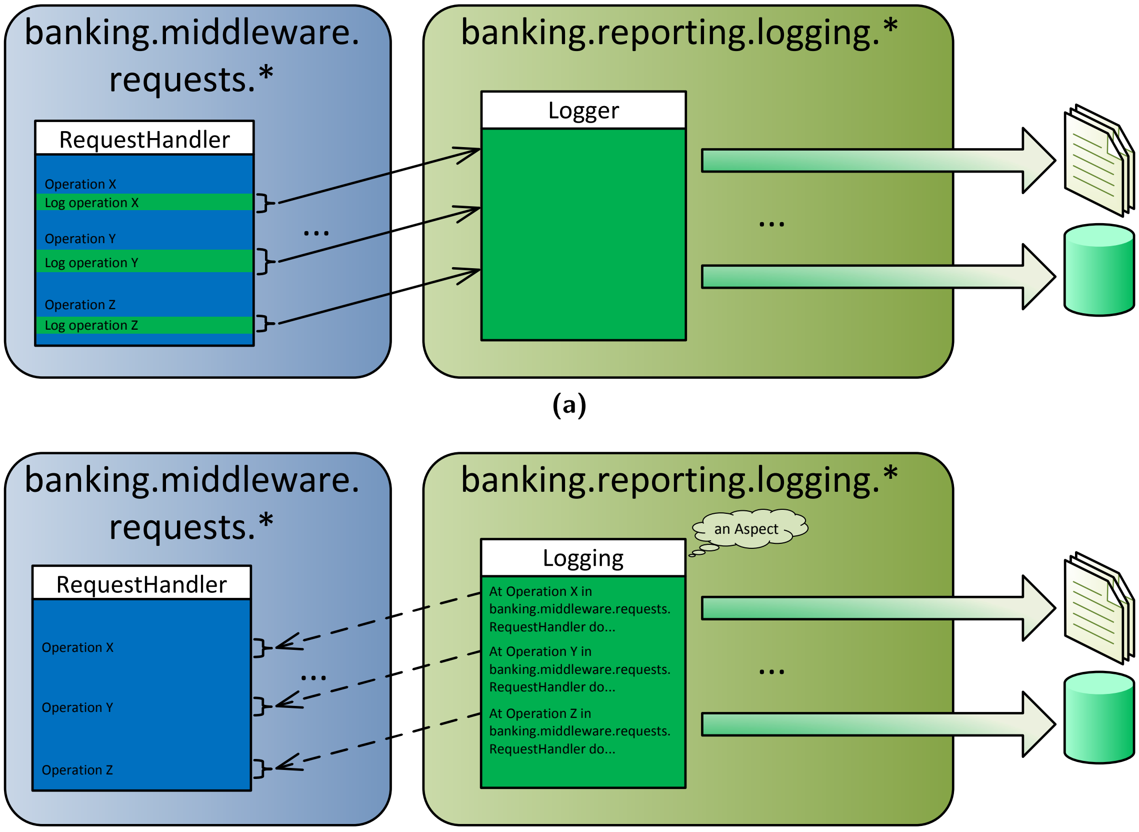
In most AOP languages aspect implementations consist of two conceptually different parts: the *functionality* code, consisting of advices, which define the behaviour that is executed upon the invocation of the aspect, and the *applicability* code, consisting of pointcuts, which determine where or when these invocations must occur. The *join point model* of the language defines the possible kinds of join points (e.g.: method or constructor calls, field references, etc.) and how they can be described by ("matched") pointcuts.

We will illustrate the difference between explicit and implicit invocation and the basic AO concepts using our running example. We assume the RequestHandler object includes three operations that need to be logged: operation X, Y and Z. Figure 4.2, on the next page, shows an object-oriented (a) and an aspect-oriented (b) implementation.

In the object-oriented implementation, each of the operations is followed by code that *explicitly invokes* (e.g.: by means of a method call) the behaviour of the Logger object and thus implements part of the logging concern within the RequestHandler object (causing scattering and tangling). In an aspect-oriented implementation no logging code is present in RequestHandler (avoiding scattering and tangling). Instead the new Logging aspect implements the logging behaviour, using *advice code*, and defines where it should be applied, by means of *pointcuts* that intercept operations X, Y and Z in RequestHandler and *implicitly invoke* the logging behaviour at those *join points*.

In conclusion, we should note that programs written in aspect-oriented languages require a special kind of compilers (or interpreters), called *weavers*. At compile-time, such weavers insert the advice code at join points, as specified by pointcuts.

[4] This principle is called *obliviousness* [20].



(a)

(b)

Figure 4.2: Explicit invocation in an OO implementation (a) and implicit invocation in an AO implementation (b) of the logging concern

4.2.4 AspectJ

AspectJ, an aspect-oriented superset of Java [94, 23], is definitely the most prominent and mature AOP language today. It was created at Xerox PARC, by a team of researchers led by Gregor Kiczales [34, 35]. The language is popular in the research community but receives major interest from mainstream software development as well [16, 50]. Further development is now organised as an open-source project led by the Eclipse Foundation [88]. The same organisation also develops the AspectJ Development Tools (AJDT) [85], which extend the Eclipse development environment with full-blown support for AspectJ.

In AspectJ, aspects can be expressed in much the same way as class definitions in Java, they can include methods, advices and pointcut definitions. Join points in AspectJ are points in the execution of a base program written in Java(/AspectJ), such as calls to or executions of methods or constructors, exception throwing, field access, etc.

As a brief demonstration of the basic features, we will show how the logging concern from our running example can be implemented using AspectJ. But first, let us look at a Java implementation of the object-oriented solution from figure 4.2a: we see that the RequestHandler class, shown in box 4.1, implements part of the logging concern by ending the methods OperationX, OperationY and OperationZ with explicit invocations (on lines 9, 15 and 21) of the logging behaviour, implemented in the Logger class (not shown).

```
1  package banking.middleware.requests;
2  import banking.reporting.logging.*;
3
4  public class RequestHandler
5  {
6     public void OperationX()
7     {
8        //...do stuff
9        Logger.logLine("Operation X was carried out!");
10    }
11
12    public void OperationY()
13    {
14       //...do stuff
15       Logger.logLine("Operation Y was carried out!");
16    }
17
18    public void OperationZ()
19    {
20       //...do stuff
21       Logger.logLine("Operation Z was carried out!");
22    }
23 }
```

Box 4.1: RequestHandler class

In an AspectJ/Java implementation of the aspect-oriented solution from figure 4.2b, we would strip the logging code from the RequestHandler class (removing lines 9, 15 and 21 from box 4.1) and combine it with an aspect like the one shown in box 4.2.

```
package banking.reporting.logging;

public aspect Logging
{
   private void logLine(String line) { /* ... */ }

   pointcut operationX() : execution(void banking.middleware.requests.
        RequestHandler.OperationX());

   after() : operationX()
   {
      logLine("Operation X was carried out");
   }

   after() : execution(void banking.middleware.requests.RequestHandler.
        OperationY())
   {
      logLine("Operation Y was carried out");
   }

   after() : execution(void banking.middleware.requests.RequestHandler.
        OperationZ())
   {
      logLine("Operation Z was carried out");
   }
}
```

Box 4.2: Logging aspect

The Logging aspect in box 4.2 implements the logging concern as a single module. It contains 3 advices that do the logging, each is triggered by a different pointcut that intercepts the execution of one of the operations that need to be logged.

AspectJ pointcuts match specific join points using a regular expression, here we see three execution pointcuts that match the execution of a method or a constructor. The pointcuts can be named or anonymous. Named pointcuts (like the one on line 7) can be referred to from one or more advices (as on line 9). Anonymous pointcuts can only be used by a single advice as they are defined directly in the header of the advice (as on lines 14 and 19).

The advices in this example are after advices, meaning they run advice code after the join point. In our case this means that operations X, Y and Z will be logged after their execution has finished. Other advice kinds supported by AspectJ include before and around. The former runs advice code before the join point and the latter runs it instead of the join point, in which case the original join point can be executed with the proceed keyword.

More information on the AspectJ language can be found in [16, 38, 50, 83, 84, 86].

4.2.5 The Fragile Pointcut Problem

The implicit invocation mechanism introduced by the aspect-orientation paradigm effectively facilitates the modularisation of crosscutting concerns. However, that does not mean that dependencies are eliminated. In fact, because pointcuts impose assumptions on the base code, implicit invocation through pointcut definitions merely reverses the direction of existing dependencies. This is the root cause of the *fragile pointcut problem*[5].

To illustrate this, we return to our logging example, as implemented in Java and AspectJ in boxes 4.1 and 4.2. We see that three explicit invocations of Logger.logLine in the RequestHandler class, are replaced by pointcuts in the Logging aspect, which are triggered by the execution of the OperationX, -Y and -Z methods of RequestHandler. As a consequence, the explicit dependencies from these methods to Logger.logLine are replaced by implicit dependencies in the opposite direction, from the pointcuts of the Logging aspect to the methods. These pointcuts impose a number of assumptions of the base code which make them *fragile* with respect to changes in the base code. For instance, our pointcuts explicitly reference the methods by their full signature. When the base code evolves and the method signatures change, the pointcuts will no longer *match* and the logging behaviour will not be invoked, which effectively breaks the functionality of the program.

Exposing detrimental dependencies caused by fragile pointcuts in aspect-oriented software designs is one of the things we hope to achieve by applying the DSM+NOV methodology.

[5] Tackling the fragile pointcut problem is one of the current hot topics in the AOSD research field [39, 69, 27, 29].

4.3 Related Work

Baldwin & Clark have a background in economy and in their publications [5, 6, 7] they take a broad view on engineering in general. Nevertheless their work has generated interest in the field of computer science, most likely due to their explicit focus on the theoretical concept of modularity, which has been a hot topic in computer science for nearly forty years.

In this section we give an overview of publications on the application of the DSM+NOV methodology of Baldwin & Clark, in the context of computer science and software development in particular. Specifically, we discuss the work of a team of researchers headed by Kevin Sullivan[6] and of Cristina Lopes and Sushil Bajracharya.

Sullivan et al.
The first application of the DSM+NOV methodology in the context of software was presented by Sullivan et al. [72] in 2001. The paper presents an analysis, using DSMs, NOV and design rules[7], of the *Key Words in Context* program, which was originally introduced by Parnas to show the importance of modularity in software [56].

The authors noted that DSMs, as used by Baldwin and Clark and in earlier work, do not allow to model the environment in which a design is embedded. To resolve this problem they introduced *Environment and Design Structure Matrices* (EDSMs), which extend the DSM concept with the notion of *environment parameters* (EPs). EPs represent entities exogenous to the system, meaning that the designer does not control them, as opposed to conventional design parameters (DPs), which are endogenous to the system.

Furthermore, Sullivan et al. reported that making justifiable estimates for the input parameters of the NOV model in the context software design remained an open challenge.

In 2005, Sullivan et al. applied the DSM+NOV methodology in the context of AOSD [73]. In that publication they proposed a new kind of information hiding interface for aspect-orientation, which abstracts crosscutting behaviour by establishing design rules which explicitly define the interface between aspects and base code entities.

Lopes & Bajracharya
In 2005, Lopes & Bajracharya published a short report [4] containing a list of open issues concerning the application of the NOV model in the context of software.

[6] Sullivan also published on applying real option theory – see Chapter 2 (2.3.6) – to software design [70, 71].

[7] Refer to Chapter 2 (2.3.3) for an explanation of the concept of design rules.

Some of the listed issues are:

- How can the numeric value from NOV be associated with meaningful attributes in software development?
- Is there a valid mapping between conventional software metrics and the parameters of NOV?
- How can differences between various forms of modular dependencies be incorporated in the model? For example, in object-oriented software, should inheritance have the same dependency *weight* as a method call?

The report concluded that a deeper understanding of the parameters in NOV, especially the link between economic and software-related properties, was needed before such questions could be answered. During our own research, as we discuss in the remainder of this chapter, we have run into many of the same questions.

Later in 2005, Lopes & Bajracharya demonstrated the use of the DSM+NOV methodology to analyse and compare OO and AO implementations of a single, real-world application [44]. Starting from an OO version they produced an AO version of the design in a step-by-step fashion. The analysis of the different evolutions of the design showed that the AO designs yielded higher NOV scores than the OO versions. This led the authors to conclude that, under the theory of modularity of Baldwin & Clark, certain AO modularisations can add value to a design.

In 2006, Lopes & Bajracharya published an updated version [45] of their 2005 paper [44]. This time they used another assumption for the technical potential parameter of the NOV model. Furthermore, to illustrate that AO modularisations are not by definition better than OO modularisations, they included an additional AO version of their running example, which was intentionally badly designed. In that case the increased usage of aspects had a detrimental effect on the NOV of the design.

This last publication is the most complete account of an effort to evaluate and compare OO and AO designs using the DSM+NOV methodology. Hence, it was the main source of inspiration for the work we present in this chapter.

4.4 Design Patterns

Design patterns are general, repeatable solutions to a commonly occurring problems in (software) design. A design pattern is not a finished design but rather a description or template stipulating how to solve a particular problem, applicable in many different situations.

The concept of design patterns was first introduced in the field of architecture and urban planning in the late 1970s [1]. In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying design patterns to software [8].

The publication of *Design Patterns – Elements of Reusable Object-Oriented Software* [22] by the *Gang-of-Four* (*GoF*) in 1994, caused the popularity of design patterns in software development to surge. This standard work provides a comprehensive catalogue of flexible object-oriented solutions to common software design problems. Each of the GoF patterns has a name that is easy to remember, a detailed, yet sufficiently abstract, problem description, and a solution expressed in terms of generalised classes and/or interfaces.

4.4.1 Aspect-Oriented Design Patterns

In 2002, Jan Hannemann & Gregor Kiczales published an article on design pattern implementations in Java and AspectJ [24, 25]. They implemented all 23 GoF patterns in both languages. In the AspectJ versions they redesigned the patterns to maximally exploit the AO features of the language. These design pattern implementations are the subject of the experiments we present in this chapter.

To evaluate their work Hannemann & Kiczales qualitatively compared the pattern implementations using 4 criteria: *locality*, *reusability*, *composability* and *(un)pluggability*. The authors concluded that in terms of these criteria AspectJ yielded a better implementation than Java for 17 of the 23 design patterns.

4.5 Experiments

In these experiments we use the DSM+NOV methodology to compare aspect-oriented and object-oriented implementations of GoF design patterns [22]. These experiments are a case study to try out the methodology and to evaluate the merits of the NOV model as a modularity metric for software design.

The design pattern implementations of Hannemann & Kiczales [24] are an excellent subject for qualitative comparison of designs, because the AspectJ and Java versions are identical in functionality but differ significantly in design.

The GoF have categorised their design patterns in three groups: Behavioural, Creational and Structural design patterns. For the sake of representativeness we conducted DSM+NOV experiments on the Java and AspectJ implementations of one pattern from each category, respectively the Observer, Builder and Composite design patterns.

Before we present the actual experiments, we provide a brief outline of the approach we followed. Then, we discuss three experiments on the Observer pattern and we formulate some provisional conclusions. Next, we present two experiments on both the Builder and the Composite pattern. Finally, we conclude this section with a summary of our findings and an evaluation of the applicability of the NOV model.

4.5.1 Approach

To explore to possibilities of the DSM+NOV methodology we devised an experimentation approach consisting of three phases. Each consecutive phase, which we call a *measurement view*, is intended to enhance the level of detail that is being measured, in search of subtle differences between the Java and the AspectJ versions. Below, we explain the different measurement views in connection with the Observer pattern experiments.

The experiments were carried out using spreadsheets generated by *DSM+NOV Tool* (see Chapter 3). The dependency values in the DSMs were filled out manually after close inspection of the source code. To define the NOV input parameters values we followed conventions and assumptions proposed by Baldwin & Clark [5] or by authors who have applied the NOV model in the context of software development (see section 4.3).

In the experiments, unless otherwise state, the NOV parameters are defined as follows:

- Following [46], the visibility cost of each i-th module, Z_i, is calculated by spreadsheet formulas based on dependency values in the DSM, refer to Chapter 3 (3.4.1);
- Following [45], the technical potential of each i-th module, σ_i, is taken as 2.5;
- The assumption for the number of design parameters in each i-th module, p_i, varies in the different measurement views (see below);
- Following [5], the complexity of each i-th module, n_i, is proportional to the number of design parameters it contains (calculated as: $n_i = \frac{p_i}{N}$, with $N = \sum_j p_j$);
- Following [46], the redesign cost of each i-th module, c_i, is taken as the relative complexity of the module (calculated as $c_i = \frac{n_i}{max_j(n_j)}$).

In all experiments the modules are source code entities on the *class-level*, corresponding to classes, interfaces and, in the case of AspectJ, aspects.

To allow every module to reach its maximal return value in the NOV computation we used the "automatic break-even points finding" feature of DSM+NOV Tool, which iteratively increases the number of simulated experiments until every module has reached its break-even point. In the spreadsheets displayed below the return values which correspond to break-even points are marked in green. Furthermore, we should note that some experiment columns in the spreadsheets are hidden to save space.

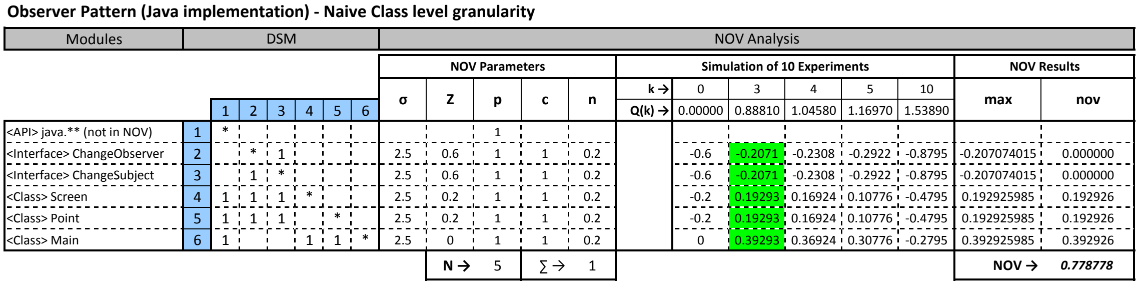
4.5.2 Observer Pattern

We conducted three experiments on the Java and AspectJ implementations of the Observer design pattern. Each experiment corresponds to a measurement view.

Naive Class-level View

Our first measurement view is the *naive class-level view*. Here, the number of design parameters of every i-th module (p_i) is taken as 1. We account for all dependencies in the source code (inheritance relations, interface implementations, type references, method calls, etc.). Furthermore, we use regular, binary DSMs, so whenever a class-level entity exhibits one or more dependencies towards another entity, the corresponding dependency value in the DSM[8] is taken as 1 and otherwise it stays 0.

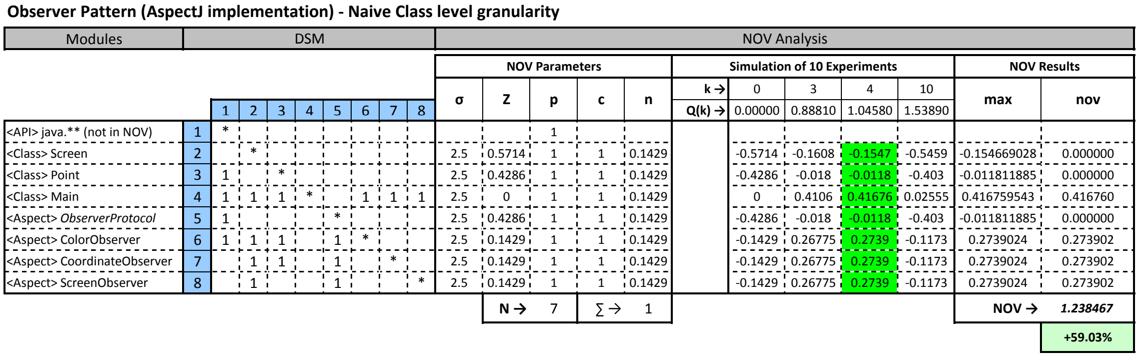
Figure 4.3 shows the DSM+NOV spreadsheets for the naive class-level views on the Java (a) and AspectJ (b) version of the Observer pattern. Using this measurement view the AspectJ version of the Observer pattern scores almost 60% better than the Java version.



Observer Pattern (Java implementation) - Naive Class level granularity

Modules	DSM							NOV Analysis												
								NOV Parameters					Simulation of 10 Experiments						NOV Results	
								σ	Z	p	c	n	k →	0	3	4	5	10	max	nov
		1	2	3	4	5	6						Q(k) →	0.00000	0.88810	1.04580	1.16970	1.53890		
<API> java.** (not in NOV)	1	*								1										
<Interface> ChangeObserver	2		*	1				2.5	0.6	1	1	0.2		-0.6	-0.2071	-0.2308	-0.2922	-0.8795	-0.207074015	0.000000
<Interface> ChangeSubject	3		1	*				2.5	0.6	1	1	0.2		-0.6	-0.2071	-0.2308	-0.2922	-0.8795	-0.207074015	0.000000
<Class> Screen	4	1	1	1	*			2.5	0.2	1	1	0.2		-0.2	0.19293	0.16924	0.10776	-0.4795	0.192925985	0.192926
<Class> Point	5	1	1	1		*		2.5	0.2	1	1	0.2		-0.2	0.19293	0.16924	0.10776	-0.4795	0.192925985	0.192926
<Class> Main	6	1			1	1	*	2.5	0	1	1	0.2		0	0.39293	0.36924	0.30776	-0.2795	0.392925985	0.392926
									N →	5	∑ →	1							NOV →	*0.778778*

(a)



Observer Pattern (AspectJ implementation) - Naive Class level granularity

Modules	DSM									NOV Analysis											
										NOV Parameters					Simulation of 10 Experiments					NOV Results	
										σ	Z	p	c	n	k →	0	3	4	10	max	nov
		1	2	3	4	5	6	7	8						Q(k) →	0.00000	0.88810	1.04580	1.53890		
<API> java.** (not in NOV)	1	*										1									
<Class> Screen	2		*							2.5	0.5714	1	1	0.1429		-0.5714	-0.1608	-0.1547	-0.5459	-0.154669028	0.000000
<Class> Point	3	1		*						2.5	0.4286	1	1	0.1429		-0.4286	-0.018	-0.0118	-0.403	-0.011811885	0.000000
<Class> Main	4	1	1	1	*		1	1	1	2.5	0	1	1	0.1429		0	0.4106	0.41676	0.02555	0.416759543	0.416760
<Aspect> *ObserverProtocol*	5	1				*				2.5	0.4286	1	1	0.1429		-0.4286	-0.018	-0.0118	-0.403	-0.011811885	0.000000
<Aspect> ColorObserver	6	1	1	1		1	*			2.5	0.1429	1	1	0.1429		-0.1429	0.26775	0.2739	-0.1173	0.2739024	0.273902
<Aspect> CoordinateObserver	7		1	1		1		*		2.5	0.1429	1	1	0.1429		-0.1429	0.26775	0.2739	-0.1173	0.2739024	0.273902
<Aspect> ScreenObserver	8		1			1			*	2.5	0.1429	1	1	0.1429		-0.1429	0.26775	0.2739	-0.1173	0.2739024	0.273902
											N →	7	∑ →	1						NOV →	*1.238467*
																					+59.03%

(b)

Figure 4.3: Naive class-level view on the Observer pattern in Java (a) and AspectJ (b)

There are a number of reasons why we consider this view to be naive. First of all, all modules are considered to be equally sized atomic entities (consisting of a single design parameter,

[8] We should note that 0s are suppressed in the DSMs in the shown spreadsheets.

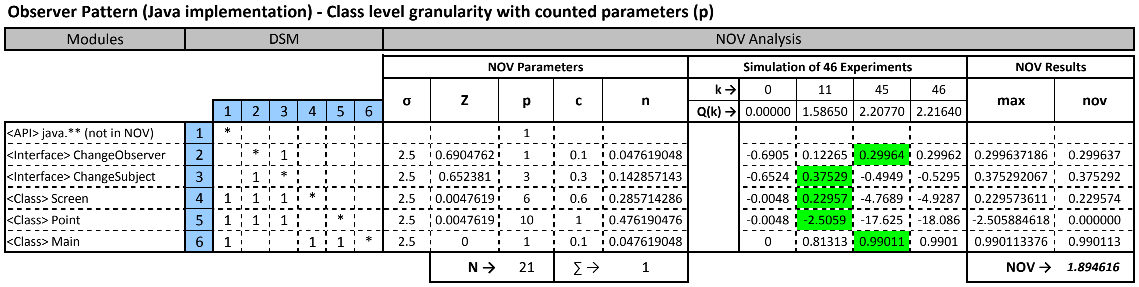
i.e. $\forall i \ : \ p_i = 1$). Consequently, all modules are equal in terms of their complexity (n) and their redesign cost (c), because the values of those parameters are based on p (see above). And secondly, because we use binary DSMs, all dependencies are treated equally as well. This ignores the fact that dependencies come in different forms (e.g.: method call vs. join point reference in a pointcut) and in different numbers (e.g.: 1 vs. 20 method calls).

Class-level View with Counted Parameters

In this second measurement view, we introduce the notion of counted design parameters as an attempt to correct the first shortcoming of the naive view. Now, the value of p_i accounts for the *size* of each i-th module. This changes the value of its complexity, n_i, and redesign cost, c_i, because the former is proportional to p_i and the latter is taken as the relative complexity of the module.

Like modules, design parameters represent parts of the studied system, but they are situated at a lower level of hierarchy, detail, abstraction or granularity[9]. We choose to take the design parameters of a module as the methods, constructors, pointcuts, advices and intertype declarations contained within the class, interface or aspect the module represents. The design patterns thus represent *parameter-level* or *method-level* source code entities, where the modules represent *module-level* or *class-level* source code entities. As the name of the measurement view implies, the p values are determined by counting the number of these parameter-level entities for each module (e.g.: the number of methods and constructors in a class).

While we believe that it is an improvement over the equally sized atomic modules of the previous measurement view, this approach is still naive in the sense that all types of parameter-level source code constructs, are treated equally (each adding 1 to p).



Observer Pattern (Java implementation) - Class level granularity with counted parameters (p)

Modules		DSM						NOV Analysis											
								NOV Parameters					Simulation of 46 Experiments					NOV Results	
								σ	Z	p	c	n	k →	0	11	45	46	max	nov
		1	2	3	4	5	6						Q(k) →	0.00000	1.58650	2.20770	2.21640		
<API> java.** (not in NOV)	1	*								1									
<Interface> ChangeObserver	2		*	1				2.5	0.6904762	1	0.1	0.047619048		-0.6905	0.12265	0.29964	0.29962	0.299637186	0.299637
<Interface> ChangeSubject	3		1	*				2.5	0.652381	3	0.3	0.142857143		-0.6524	0.37529	-0.4949	-0.5295	0.375292067	0.375292
<Class> Screen	4	1	1	1	*			2.5	0.0047619	6	0.6	0.285714286		-0.0048	0.22957	-4.7689	-4.9287	0.229573611	0.229574
<Class> Point	5	1	1	1		*		2.5	0.0047619	10	1	0.476190476		-0.0048	-2.5059	-17.625	-18.086	-2.505884618	0.000000
<Class> Main	6	1			1	1	*	2.5	0	1	0.1	0.047619048		0	0.81313	0.99011	0.9901	0.990113376	0.990113
									N →	21	∑ →	1						NOV →	*1.894616*

Figure 4.4: Class-level view with counted parameters for the Observer pattern in Java

The result for the Java version of the Observer pattern is shown in figure 4.4. The new measurement approach results in increased NOV scores for both versions. However, the NOV of the AspectJ version increased more and is now almost 125% higher compared to the Java version.

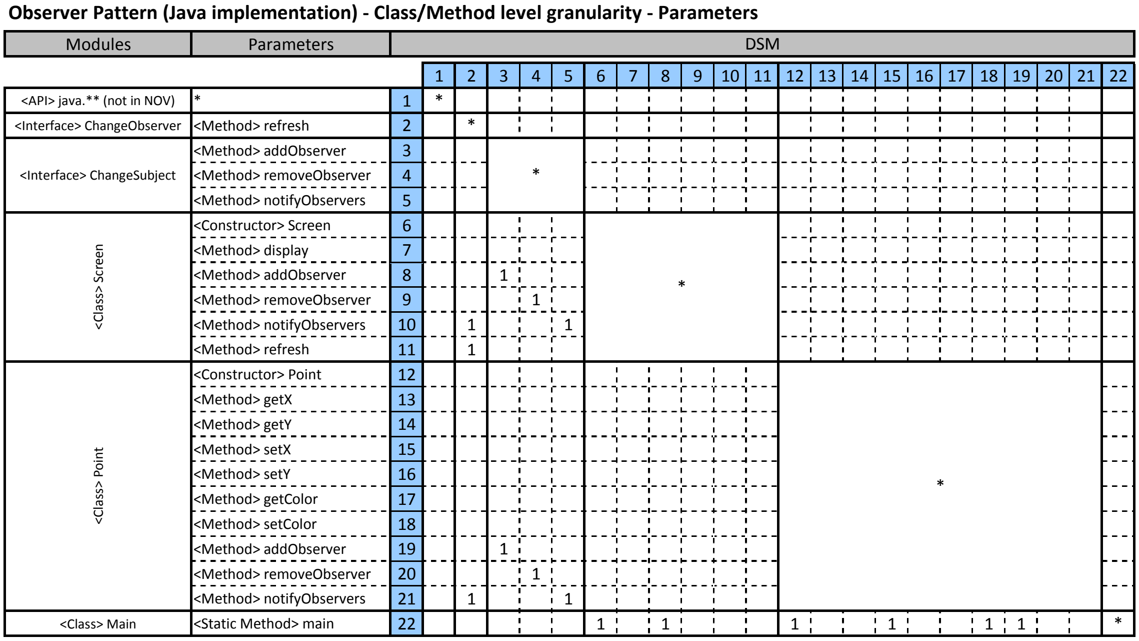
[9] Refer to Chapter 3 (3.4.2).

Class/Method-level View

In the previous measurement views we used binary values to account for dependencies among modules. We thereby ignored the fact that the dependencies in the design pattern implementations come in different forms and in different numbers. Furthermore, we ignored the subtleties of dependencies among parameter-level source code constructs (e.g.: from a pointcut definition in an aspect to a method in a class) by summarising them to binary dependency values per ordered pair of class-level modules.

In this *class/method-level view*, we attempt to increase the level of measured detail. To achieve this *finer granularity*, we introduce two changes. As a first change, we want to express dependencies among modules as numerical, instead of binary, values – by using a numerical DSM (NDSM) – to account for different *numbers* of dependencies. As a second change, we want to explicitly document parameter-level dependencies in a separate DSM diagram, which zooms in on the contents of modules. We first look at the second change.

We call the separate DSM the *parameter-level* or *method-level* DSM and we use it to confront individual methods, pointcuts, advices, etc. As an example, figure 4.5 shows the method-level DSM for the Java version of the Observer pattern.



Observer Pattern (Java implementation) - Class/Method level granularity - Parameters

Modules	Parameters		DSM																					
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
<API> java.** (not in NOV)	*	1	*																					
<Interface> ChangeObserver	<Method> refresh	2		*																				
<Interface> ChangeSubject	<Method> addObserver	3			*																			
	<Method> removeObserver	4																						
	<Method> notifyObservers	5																						
<Class> Screen	<Constructor> Screen	6						*																
	<Method> display	7																						
	<Method> addObserver	8			1																			
	<Method> removeObserver	9				1																		
	<Method> notifyObservers	10		1			1																	
	<Method> refresh	11		1																				
<Class> Point	<Constructor> Point	12												*										
	<Method> getX	13																						
	<Method> getY	14																						
	<Method> setX	15																						
	<Method> setY	16																						
	<Method> getColor	17																						
	<Method> setColor	18																						
	<Method> addObserver	19			1																			
	<Method> removeObserver	20				1																		
	<Method> notifyObservers	21		1			1																	
<Class> Main	<Static Method> main	22						1		1				1			1			1	1			*

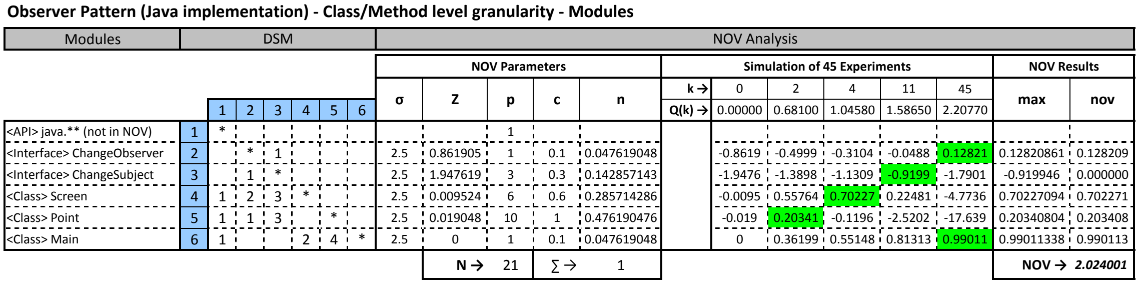
Figure 4.5: Method/Parameter-level DSM for the Observer pattern implemented in Java

To create such DSMs in a spreadsheet, we added a specialised feature of DSM+NOV Tool[10] which generates a parameter-level DSM with the right number of parameters per module, based on the *p* values specified in an existing, *module-level* DSM+NOV spreadsheet.

[10] Refer to Chapter 3 (3.4.2).

As far as the DSM are concerned, this clearly allows us to study dependencies in more detail. However, because the NOV model is only designed to work at the module-level we cannot calculate NOV scores based on parameter-level DSMs. Therefore we let DSM+NOV Tool link the parameter-level DSM to the original module-level DSM by summarising through summing. This means that the dependency between an ordered pair of modules, as documented in the module-level DSM, is now being calculated by summing the dependencies between the parameters of those modules, increased with any module-level dependencies that were already filled out in the module-level DSM[11].

This way the NOV calculation stays at the module-level, while still taking into account information from the parameter-level DSM. The result for the Java version of the Observer pattern is shown in figure 4.6.



Observer Pattern (Java implementation) - Class/Method level granularity - Modules

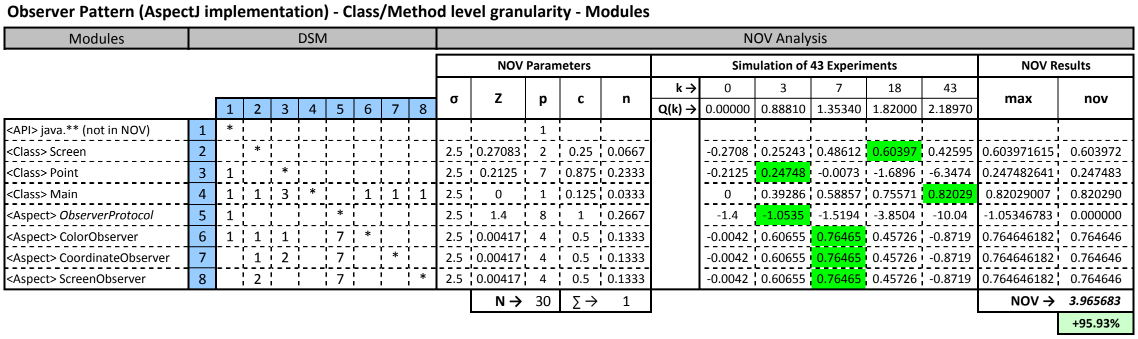
Modules	DSM							NOV Analysis												
								NOV Parameters					Simulation of 45 Experiments						NOV Results	
								σ	Z	p	c	n	k →	0	2	4	11	45	max	nov
		1	2	3	4	5	6						Q(k) →	0.00000	0.68100	1.04580	1.58650	2.20770		
<API> java.** (not in NOV)	1	*								1										
<Interface> ChangeObserver	2		*	1				2.5	0.861905	1	0.1	0.047619048		-0.8619	-0.4999	-0.3104	-0.0488	0.12821	0.12820861	0.128209
<Interface> ChangeSubject	3		1	*				2.5	1.947619	3	0.3	0.142857143		-1.9476	-1.3898	-1.1309	-0.9199	-1.7901	-0.919946	0.000000
<Class> Screen	4	1	2	3	*			2.5	0.009524	6	0.6	0.285714286		-0.0095	0.55764	0.70227	0.22481	-4.7736	0.70227094	0.702271
<Class> Point	5	1	1	3		*		2.5	0.019048	10	1	0.476190476		-0.019	0.20341	-0.1196	-2.5202	-17.639	0.20340804	0.203408
<Class> Main	6	1			2	4	*	2.5	0	1	0.1	0.047619048		0	0.36199	0.55148	0.81313	0.99011	0.99011338	0.990113
									N →	21	∑ →	1							NOV →	*2.024001*

Figure 4.6: Class/Method-level measurement view for the Java version of the Observer pattern: Class/Module-level DSM (+NOV calculation) linked to the method/parameter-level DSM in figure 4.5 through summarising by summing

The beauty of this solution is that it also implements the first change we wanted to introduce. By linking both DSMs by summing parameter-level dependencies, the module-level DSM now becomes an NDSM which literally counts the *number* of dependencies (e.g.: 20 calls from methods of class A to methods of class B result in a dependency of 20 from A to B). However, we should note this measurement view still does not differentiate between forms of dependencies (e.g.: a method calls still have the same "weight", namely 1, as inheritance relations).

We expected that the Class/Method-level view would have a substantial effect on NOV outcomes. For instance, we thought this would view would penalise the fragile enumeration pointcuts in the AspectJ implementation of the pattern. But the results for the Observer pattern contradict this: the AspectJ version still scores 95% better than the Java version, as shown by figure 4.7 on the next page. Compared to the previous measurement view the NOV of the AspectJ version decreases by 7%, while the score of the Java version increases by 7%.

[11] To account for strict module-level/class-level dependencies such as inheritance, see Chapter 3 (3.4.2).



Observer Pattern (AspectJ implementation) - Class/Method level granularity - Modules

Modules		DSM								NOV Analysis												
										NOV Parameters					Simulation of 43 Experiments						NOV Results	
										σ	Z	p	c	n	k →	0	3	7	18	43	max	nov
		1	2	3	4	5	6	7	8						Q(k) →	0.00000	0.88810	1.35340	1.82000	2.18970		
<API> java.** (not in NOV)	1	*										1										
<Class> Screen	2		*							2.5	0.27083	2	0.25	0.0667		-0.2708	0.25243	0.48612	0.60397	0.42595	0.603971615	0.603972
<Class> Point	3	1		*						2.5	0.2125	7	0.875	0.2333		-0.2125	0.24748	-0.0073	-1.6896	-6.3474	0.247482641	0.247483
<Class> Main	4	1	1	3	*		1	1	1	2.5	0	1	0.125	0.0333		0	0.39286	0.58857	0.75571	0.82029	0.82029007	0.820290
<Aspect> *ObserverProtocol*	5	1				*				2.5	1.4	8	1	0.2667		-1.4	-1.0535	-1.5194	-3.8504	-10.04	-1.05346783	0.000000
<Aspect> ColorObserver	6	1	1	1		7	*			2.5	0.00417	4	0.5	0.1333		-0.0042	0.60655	0.76465	0.45726	-0.8719	0.764646182	0.764646
<Aspect> CoordinateObserver	7		1	2		7		*		2.5	0.00417	4	0.5	0.1333		-0.0042	0.60655	0.76465	0.45726	-0.8719	0.764646182	0.764646
<Aspect> ScreenObserver	8		2			7			*	2.5	0.00417	4	0.5	0.1333		-0.0042	0.60655	0.76465	0.45726	-0.8719	0.764646182	0.764646
											N →	30	∑ →	1							**NOV →**	***3.965683***
																						+95.93%

Figure 4.7: Class/Method-level measurement view for the AspectJ version of the Observer pattern: Class/Module-level DSM (+NOV calculation)

Summary

Figure 4.8 summarises our findings for the Observer pattern, it lists the three experiments – based on the three measurement views – we discussed above and a fourth experiment we discuss below.

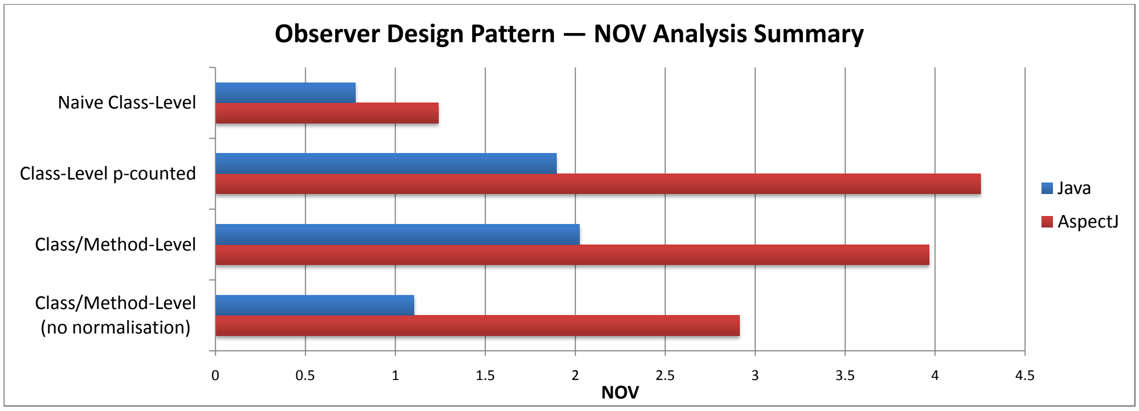


Figure 4.8: NOV measurement results for the Observer pattern

4.5.3 Provisional Conclusions

Considering the results of the last two experiments on the Observer pattern implementations, it is clear that significantly higher dependency values only have a minor decreasing effect on NOV scores. Before we discuss experiments on other design patterns this observation requires further investigation.

Even when dependency values for a module are doubled or even tripled NOV scores are only slightly affected and sometimes not affected at all. Our first suspicion was that this could be due to the fact that our implementation of the NOV model normalises the negative nov_i

values to 0. So although high dependencies will increase the visibility cost (Z) of some of the modules, this effect is essentially neutralised as soon as the return values are pulled below 0.

While this is true, rerunning the third experiment on the Observer pattern without the normalisation step, showed that it is not the (only) reason. Even though the AspectJ version (figure 4.7) has significantly higher dependency values than the Java version (figure 4.6), skipping the normalisation step had a much larger relative decreasing effect on the NOV score for the latter, as shown by figure 4.8 on the previous page.

We suspect that this problem is related to the fact that we combine NOV with NDSMs. To the best of our knowledge, that has not been done before – not by Baldwin & Clark nor by others – which leads us to assume that the NOV model was not designed for that purpose[12].

Because of the unhappy marriage between NDSMs and NOV, we have refrained from introducing a fourth measurement view which makes a distinction between forms of dependencies (a shortcoming mentioned before). Furthermore, in the experiments on other design patterns, which we discuss below, we restrict ourselves to the class-level view with counted parameters. Instead we experiment with alternative assumptions for NOV input parameters.

4.5.4 Builder Pattern

We conducted two experiments on the Java and AspectJ implementations of the Builder design pattern. In each experiment we used the class-level measurement view with counted parameters. In the second experiment we used an alternative assumption for the redesign cost parameter (c): following [5], the value of the parameter was fixed at 1 for every module. Until now this parameter was taken as the relative complexity of the module (following [46]).

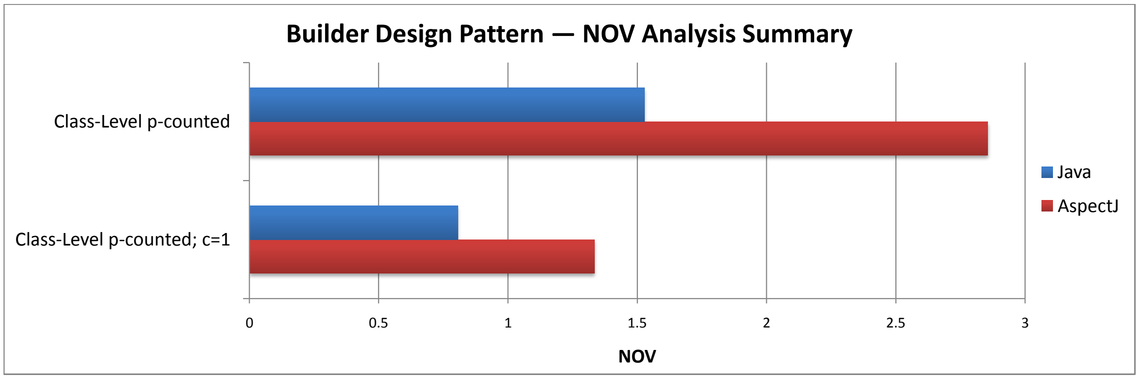


Figure 4.9: NOV measurement results for the Builder pattern

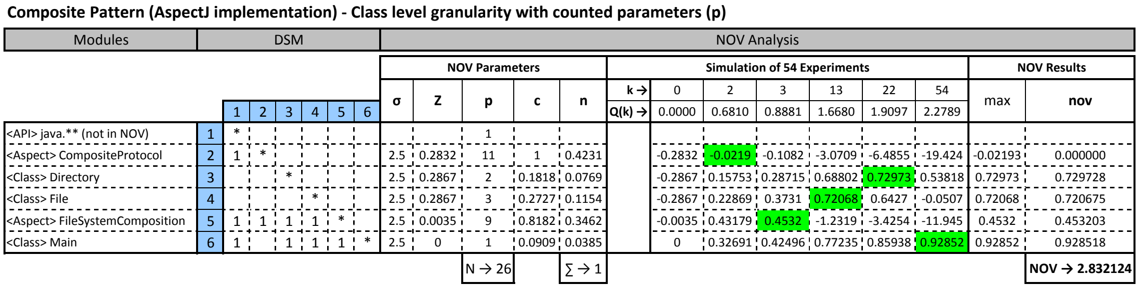
[12] As noted earlier in Chapter 2 (2.3.6) and Chapter 3 (3.4.2).

Figure 4.9, on the previous page, summarises the results for both experiments. Again, the AspectJ implementation scores much higher than the Java equivalent. The effect of the alternative assumption for the redesign cost is a decrease in NOV for both versions. Although the AspectJ score is relatively more affected, it remains 65% above the score of the Java version.

4.5.5 Composite Pattern

Our final two DSM+NOV experiments were carried out on the Java and AspectJ implementations of the Composite design pattern. In both experiments we used the class-level measurement view with counted parameters. In the second experiment we introduce a novel assumption for the complexity parameter (n).

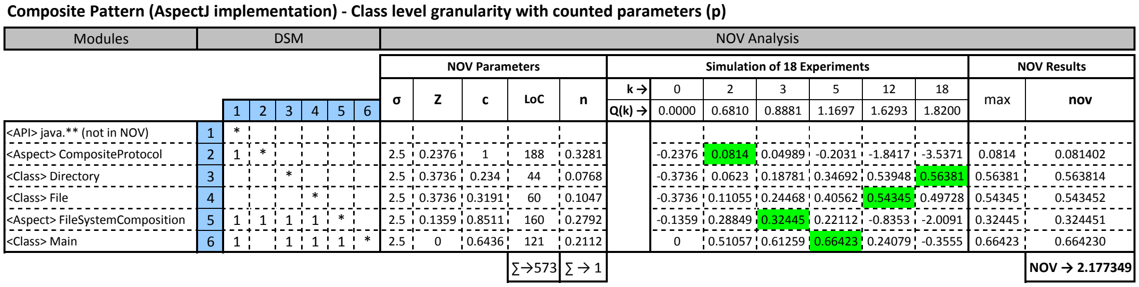
Until now, the complexity of a module was proportional to the number of design parameters the module contains (following [5]). Because we consider the number of methods, constructors, pointcuts, advices, etc. to be a rather inaccurate estimate of the actual size of classes, interfaces or aspects, let alone of their complexity, we try out an alternative approach. We propose to take the complexity of a module as the proportional amount of lines of code (LoC) contributed by the source code entity the module represents, relative to the total number of lines of the studied software system. Formally the complexity of the i-th module is computed as $n_i = \frac{LoC_i}{\sum\limits_j LoC_j}$.



Composite Pattern (AspectJ implementation) - Class level granularity with counted parameters (p)

Modules		DSM						NOV Analysis													
								NOV Parameters					Simulation of 54 Experiments							NOV Results	
								σ	Z	p	c	n	k →	0	2	3	13	22	54	max	nov
		1	2	3	4	5	6						Q(k) →	0.0000	0.6810	0.8881	1.6680	1.9097	2.2789		
<API> java.** (not in NOV)	1	*								1											
<Aspect> CompositeProtocol	2	1	*					2.5	0.2832	11	1	0.4231		-0.2832	-0.0219	-0.1082	-3.0709	-6.4855	-19.424	-0.02193	0.000000
<Class> Directory	3			*				2.5	0.2867	2	0.1818	0.0769		-0.2867	0.15753	0.28715	0.68802	0.72973	0.53818	0.72973	0.729728
<Class> File	4				*			2.5	0.2867	3	0.2727	0.1154		-0.2867	0.22869	0.3731	0.72068	0.6427	-0.0507	0.72068	0.720675
<Aspect> FileSystemComposition	5	1	1	1	1	*		2.5	0.0035	9	0.8182	0.3462		-0.0035	0.43179	0.4532	-1.2319	-3.4254	-11.945	0.4532	0.453203
<Class> Main	6	1		1	1	1	*	2.5	0	1	0.0909	0.0385		0	0.32691	0.42496	0.77235	0.85938	0.92852	0.92852	0.928518
										N → 26		∑ → 1									NOV → 2.832124

(a)



Composite Pattern (AspectJ implementation) - Class level granularity with counted parameters (p)

Modules		DSM						NOV Analysis													
								NOV Parameters					Simulation of 18 Experiments							NOV Results	
								σ	Z	c	LoC	n	k →	0	2	3	5	12	18	max	nov
		1	2	3	4	5	6						Q(k) →	0.0000	0.6810	0.8881	1.1697	1.6293	1.8200		
<API> java.** (not in NOV)	1	*																			
<Aspect> CompositeProtocol	2	1	*					2.5	0.2376	1	188	0.3281		-0.2376	0.0814	0.04989	-0.2031	-1.8417	-3.5371	0.0814	0.081402
<Class> Directory	3			*				2.5	0.3736	0.234	44	0.0768		-0.3736	0.0623	0.18781	0.34692	0.53948	0.56381	0.56381	0.563814
<Class> File	4				*			2.5	0.3736	0.3191	60	0.1047		-0.3736	0.11055	0.24468	0.40562	0.54345	0.49728	0.54345	0.543452
<Aspect> FileSystemComposition	5	1	1	1	1	*		2.5	0.1359	0.8511	160	0.2792		-0.1359	0.28849	0.32445	0.22112	-0.8353	-2.0091	0.32445	0.324451
<Class> Main	6	1		1	1	1	*	2.5	0	0.6436	121	0.2112		0	0.51057	0.61259	0.66423	0.24079	-0.3555	0.66423	0.664230
											∑→573	∑ → 1									NOV → 2.177349

(b)

Figure 4.10: Class-level view with counted parameters for the Composite pattern in AspectJ; using the conventional (a) and a LoC-based (b) assumption for module complexity

Figure 4.10, on the previous page, shows DSM+NOV spreadsheets for the AspectJ version of the Composite design pattern, using the conventional (a) and the new (b) assumption for the complexity parameter.

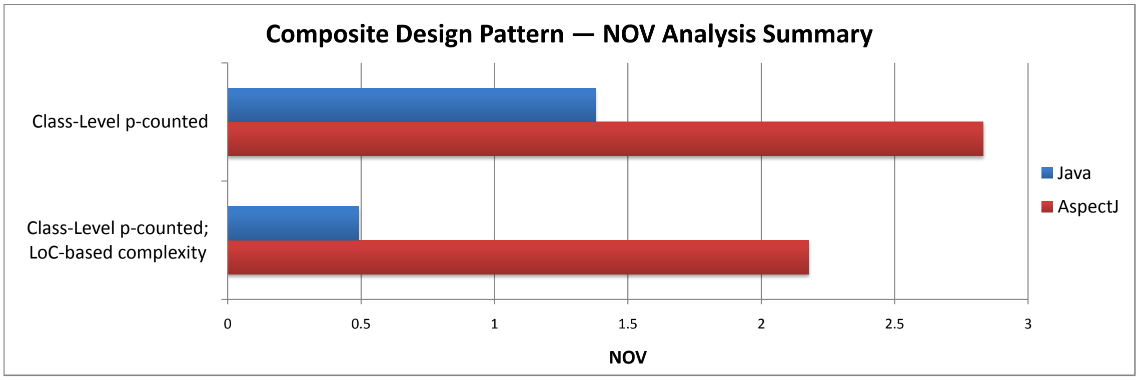


Figure 4.11: NOV measurement results for the Composite pattern

Figure 4.11 summarises the results for the Composite pattern experiments. The AspectJ version of the pattern scores higher in both experiments. The effect of the LoC-based complexity is remarkable. The new assumption results in a decrease in NOV for both versions, but the relative (and absolute) difference is much larger for the Java version in comparison with AspectJ version.

4.5.6 Conclusions

4.5.6.1 Summary

We observed that experimenting with different measurement approaches – or *views* as we call them – and alternative NOV parameter assumptions, had an influence on the measured NOV scores but never significantly affected the dominance of the AspectJ versions over the Java versions of the studied design patterns.

However, we should stress that, due to a number of issues regarding the applicability of the NOV model as a software metric (see below), the results of our experiments cannot be used to draw conclusions on the claimed modularity advantages of the aspect-oriented design pattern implementations, over the object-oriented implementations.

4.5.6.2 Evaluating the Applicability of the NOV Model

Based on the experiments and findings discussed above we now present our conclusions with regard to the applicability of NOV model as a modularity metric for software design.

As already noted in Chapter 2, the NOV model is not a black box. Its application for a particular purpose requires a multitude of assumptions, appropriate for the given context, about the input parameters of the model. This requires a thorough understanding of the meaning of those parameters.

To explore the possibilities of the DSM+NOV methodology, we experimented with alternative measurement approaches and assumptions for the input parameters of the NOV model. However, we were unable to find an optimal approach.

Our experiment with the LoC-based assumption for the complexity parameter is an illustrative example. Even though all other parameters were kept constant, the results show a striking discrepancy between the effect on the NOV score of the AspectJ version and the effect on the result for the Java version of the Composite pattern. While we thought that LoC-based module complexity makes sense in the context of software, we have most likely misunderstood the true meaning of the parameter.

We consider the idea to assess modularity using a *module-level* and a *parameter-level* DSM to be a powerful concept. Moreover, we hoped this approach would reveal more about the interactions between design pattern participants, for instance with regard to fragile pointcuts. However the results were disappointing. Due to the problematic combination of NDSM and the NOV model, significantly higher (numerical) dependency values only had a negligible decreasing effect on the NOV scores. We do not rule out the possibility that further adjustments to the input parameters of the NOV model could render it more suitable to be combined with numerical dependency values. However, our insight into the meaning of these parameters in the context of software is insufficient to fine-tune the model ourselves.

We must conclude that the current level of understanding of the input parameters and the outcome of the NOV model is limited and does not warrant the application of the model as a modularity metric for real-world software designs.

We are not alone in these observations. Other authors have expressed similar concerns, as noted in our discussion of related work in section 4.3. In fact, all previous publications on the application of the NOV model to software design, mention at least some reservations about the applicability of the model. For example, in [72] Sullivan et al. said:

> "[Quantitative models] need not be perfect. However, they must capture the most important terms and their assumptions and operation must be known and understood so that analysts understand and can evaluate their predictions."

In our opinion, these requirements are not met by the current state of the NOV model in software. Furthermore, our findings re-establish the question as to whether the model is at all applicable in this context.

Possibly, developing a new quantitative model based on DSMs, specifically designed to assess modularity in software, would be a better approach for future research than further attempts to grasp and adapt the NOV model. However, that is outside the scope of this dissertation.

4.6 Conclusions

In this chapter we have presented an evaluation the DSM+NOV methodology for quantitative assessment of modularity in software designs.

To experiment with the methodology, we applied it to aspect-oriented and object-oriented implementations of the GoF design patterns, in order to assess the claimed modularisation advantages of the former implementations. However, we first provided an extensive introduction to the novel aspect-orientation paradigm.

In our experiments we presented three *measurement views* and we conducted an initial exploration of the value space of the input parameters of NOV model. The class/method-level measurement view and the LoC-based assumption for the complexity parameter, which we introduced here, constitute novel ideas in the field.

Based on our observations, we have concluded that we lack a solid rationale to support application of the NOV model in the context of software development. Hence, the results of our assessment cannot be used to draw conclusions with regard to the modularisation properties of the aspect-oriented and object-oriented design pattern implementations.

Despite the issues of the NOV model, our experiments suggest that DSM diagrams alone are indeed a powerful way to document and visualise dependencies in software implementations. This is confirmed by our research, which we present in the following two chapters, into to the application of DSM visualisations in support tools for software development.

Chapter 5

DSM-based Code Browsers

This chapter covers our research into DSM-based support tools for object-oriented software development (OOSD). We argue that DSMs are well-suited to be applied to OOSD and offer a novel approach to dependency management. We propose a rationale for an extendable DSM-based source-code browser with analysis capabilities based on metaprogramming. Subsequently, we present a prototype of such a tool. In the next chapter we demonstrate how this prototype can assist developers in common tasks.

5.1 Introduction

In Chapter 2 we introduced the Design Structure Matrix (DSM) as a type of diagram that focuses on the modular structure of systems and the dependencies among their constituent parameters, parts or modules. In our opinion these characteristics make DSM diagrams a useful instrument to support object-oriented software development (OOSD). Specifically, we believe that DSMs are suitable to visualise the hierarchical structure of software components and to help developers with managing the different kinds of dependencies among those[1]. The work we present in this chapter attempts to validate this assumption by extending source-code browsers with DSM capabilities.

Source-code browsers are tools that integrate with development environments to extend them with facilities that enable easy navigation through the implementation of a piece of software, usually by means of an abstract representation that is coupled to the actual constructs in the source code. By keeping the representation, which is commonly based on some kind of diagrams, in sync with the implementation at all times, both are said to co-evolve [17, 76]. In case of DSM-based code browsers, co-evolution means there is a causal link between changes in the source code and resulting shifts in the dependency patterns visualised by a DSM diagram.

[1] As explained in Chapter 4 (4.5.6.2), we think that the Net Option Value (NOV) model currently cannot be confidently applied to study software designs, therefore we focus exclusively on DSMs in this chapter.

Although code browsers are metaprograms[2] in their own right, exposing metaprogramming facilities to the user can make them significantly more powerful. Such facilities enable developers to obtain diverse kinds of knowledge about a software implementation by writing small, ad-hoc analysis programs that reason about the source code using a framework of abstractions provided by the tool. In a DSM-based tool such metaprograms could use the abstractions – representing modules and dependencies in the source code – that make up the DSM, as a new reflective application programming interface (API) to reason about the source code.

In short, the research we discuss here combines design structure matrixes, source-code browsers and metaprogramming. To investigate the opportunities this combination holds for OOSD, we have formulated a rationale and created a prototype of an extendable DSM-based source-code browser with analysis capabilities based on metaprogramming. This prototype, or future similar tools, can both passively and actively support developers in acquiring important information, for instance with regard to decisions concerning local architectural restructurings. This is demonstrated by the case studies we present in Chapter 6.

In section 5.2 of this chapter, we discuss some observations about the discipline of OOSD that make the case for DSM-based support tools. Next, section 5.3 presents a rationale for such a tool consisting of a set of requirements and the approach we propose in the form of a DSM-based source-code browser. After that, section 5.4 presents *DSMBrowser*, our prototype implementation. Finally, we discuss some related work in section 5.5 and we conclude this chapter in section 5.6.

5.2 Motivation

In this section we deal with the motivations for our research into DSM-based support tools for OOSD. First, we discuss modularisation and dependencies in object-oriented (OO) software and we present of an argument for a hierarchical interpretation of those phenomena. Next, we present arguments that stress the importance of dependency management in OOSD.

5.2.1 Modularity and Dependencies in Object-Oriented Software

In object-orientation, today's de-facto software development paradigm, software is constructed from cooperating entities called objects. Objects are data structures with behaviour – they contain data fields and units of behaviour called methods. In most object-oriented languages objects are instances of classes, which act as static blueprints. Objects must co-operate to perform the functions of the software. Therefore, an object can hold references to other objects in its data fields and its methods can access fields of other objects or invoke

[2] Metaprograms are commonly defined as programs that reason about or manipulate other programs (or themselves) as their data.

external methods by sending messages to other objects. Objects can also engage in other relationships with one another. For instance, objects can be related through inheritance.

These relationships bring about different kinds of dependencies among objects and their methods. Both objects (or their classes) and methods can be seen as modules – in the sense of the definitions we discussed in Chapter 2 (2.3.1) – on different hierarchical levels.

Object-orientation provides technical mechanisms that create a grouping hierarchy – in parallel with an inheritance hierarchy – of modular abstractions. Methods represent portions of code as abstract units of behaviour and are at the bottom of this hierarchy. One level up we find objects which group methods, together with data structures, into abstract modules of data and behaviour. Objects, for their part, can be arranged in packages, bundles, namespaces, etc. Such groups of objects do not always correspond to functional abstractions, but nevertheless these techniques provide an additional level – or multiple levels, as these groupings can usually be nested – in the hierarchy.

Ideally, the modularisation that is achieved using such technical mechanisms, groups conceptually related and separates conceptually unrelated things. In other words, the technical modularisation should align with a conceptual modularisation – one that makes sense with regard to the requirements for the system in question. However, such correspondence is not always trivial to achieve.

Modularisations in software are never one-dimensional, but span multiple hierarchical levels. Hence, dependencies among modules can cross the boundaries of surrounding modules on higher levels. For example, a method might invoke a method of the same object (no boundary crossed), of another object in the same package (1 boundary crossed) or of an object in another package (2 boundaries crossed). Furthermore, dependencies can intersect with hierarchical levels when a module depends on a module situated on a different level. For instance, in a statically typed language a method might reference a class by its name as the type of a variable. Moreover, dependencies on a particular level cannot be ignored on higher levels in the hierarchy. For example, inheritance between objects in different packages introduces a dependency among those objects but also between the involved packages.

In summary, both modularity and dependency in object-oriented software should be treated as hierarchical phenomena.

5.2.2 Dependency Management

While object-oriented modularisation has its limitations[3], it simplifies the design of modular architectures compared to older paradigms. Such architectures facilitate the cooperation of

[3] In Chapter 4 we discuss Aspect-Oriented Software Development (AOSD), a new paradigm that is intended to resolve important limitations of object-oriented modularisation.

large groups of developers and can thus increase the scalability of software development. However, a modular architecture does not mean that dependencies among software components can be ignored. In fact, there are several arguments that stress the importance of true dependency management for OOSD.

Coordination efforts

The implementation of a large software system is rarely the work of one person or a small team, but usually requires the cooperation of large groups of people with diverse specialisations and responsibilities.

Whenever multiple developers cooperate, implementation-level dependencies translate to dependencies among individuals and groups of people. For example, when implementation components that depend on each other are the responsibility of different people, certain coordination efforts are essential to guarantee their proper functioning. Such coordination takes many forms, ranging from verbal agreements among individual programmers to more advanced methods. The cost related to coordination efforts in large-scale projects can be significant [40].

Existing and Third-party Code

The source code that makes up large software implementations is unlikely to be entirely written from scratch. Existing or third-party written code is used in virtually all software development projects, even for new applications that do not extend or improve on an earlier version. This code takes very diverse forms – libraries, frameworks, middleware components, etc. and recycled code in general – and originates from several sources both in- and outside a company – it may be written in-house by the same team or by another team, development may be out-sourced to employees of a consultancy firm (on-site or elsewhere, possibly even abroad) or it can be bought as a product from another company.

Existing and third-party written code further complicate dependencies in software implementations and thus affect the corresponding dependencies among people as well.

The use of existing code extends dependencies among people over time. For instance, when old, “legacy” code is used, crucial knowledge might have been lost because the responsible developers may have left the company. This hinders the maintenance and evolution of the software system. In some cases, reengineering or even entirely replacing legacy components, is the only solution. However, such steps cannot be taken overnight, especially in mission critical applications. They require a meticulous planning and a design which takes all dependencies, from components that are to stay to the legacy components, into account.

The use of third-party written code extends the range of involved parties – from individual developers and teams to departments and even whole companies – and therefore it both increases the number and widens the scope of dependencies among people. This can be a source of problems, for instance when the involved parties do not share the same interests. This is especially the case when dependencies involve products from different companies – each protecting its own investments and market shares. A consequence is that companies usually do not have the same amount of control over code that is supplied by other firms than over what is developed in-house. For example, it may be unavoidable that a new version of a third-party framework introduces changes that are incompatible with some applications that depend on it. In that sense, third-party code can be a moving target.

The Importance of Dependency Management
In summary, it is of vital importance that developers know well what their code relies on and how and why it does so. This emphasises the importance of dependency management for (object-oriented) software development. Sound dependency management supports co-ordination and communication among involved parties. It requires that efforts are made to know and to keep track of dependencies, throughout the lifecycle of a software product. Moreover, it means that opportunities for improved modularisation – resulting in dependency minimisations – are detected, evaluated and pursued at various development stages.

5.3 Rationale

In this section we present a rationale for a novel support tool for OOSD based on DSMs. We first list a number of requirements which follow from the motivations discussed above. Next, we propose an approach in the form of an extendable DSM-based source-code browser with analysis capabilities based on metaprogramming.

5.3.1 Requirements

The central goal is to devise a tool that supports dependency management and integrates well with the day-to-day practice of object-oriented software development. We infer five requirements for such a tool from the motivations we discussed earlier:

Requirement 1 It is of key importance to offer a visual overview of the dependencies among different modules in a software implementation. The visualisation should enable users to study modularisations, for instance to verify whether technical modularisations map to conceptual ones and whether design choices are upheld. The tool should distinguish different dependency kinds on different hierarchical levels. In other words, it should explicitly deal with the hierarchical nature of both modularity and dependencies in OO software.

Requirement 2 Furthermore, the tool should be flexible enough to enable developers to focus on the regions and hierarchical levels of their choice, instead of burying them with too much information in an overloaded visualisation.

Requirement 3 The provided visualisation should not be static, on the contrary it should at least support interactive navigation, both laterally and hierarchically. Additionally, visualisation elements on every level should be linked to the underlying source code constructs, allowing users to move back and forth between the visualisation and the source code.

Requirement 4 The tool should help developers to find and evaluate modularisation opportunities in the form of local architectural restructurings. We do not seek an instrument that autonomously introduces such restructurings, nor a general-purpose expert system. Rather, the tool should offer a set of customisable, manually-triggered analysis features that provide information and indicative suggestions – with regard to the minimisation of dependencies through object-oriented modularisations – that support, rather than replace, human decisions.

Requirement 5 Finally, the tool should be extendable so that users can customise it for specific applications and for deployment in specific environments.

5.3.2 Approach

The solution we propose to satisfy the requirements listed above is an extendable DSM-based source-code browser with analysis capabilities based on metaprogramming. We now discuss the prominent aspects of this approach.

5.3.2.1 DSM-based Visualisation

We believe that DSMs, through their focus on modular structures and dependencies, offer a suitable basis for the visualisation described by requirements 1 and 2.

While some modules in object-orientation, such as objects or methods, are apparent as explicit units of code, this is not always true for others, such as packages or namespaces. Most dependencies in object-orientation are explicitly defined, but often by individual lines of code (e.g.: a method call or the specification of inheritance relations in a class header), which makes them hard to track down. However, a DSM-based visualisation can display all modules in the same, explicit, way and can offer a convenient overview of all dependencies, no matter where or how they are defined.

We suggest to differentiate types of dependencies by labelling them with different "weights". Additionally, the DSMs should accurately represent the number of dependencies between modules. Therefore we propose to use NDSMs[4], instead of regular, binary, DSMs. In the remainder of this chapter we will use the term "DSM" to refer to NDSMs.

[4] NDSMS are DSMs with numerical, instead of binary, dependency values, see Chapter 2 (2.2). Unlike in Chapter 4 (4.5.3), their use can not pose a problem here as they are not combined with NOV analysis.

To comply with requirement 1 the visualisation should explicitly deal with the hierarchical nature of modularity and dependencies in object-orientation. However, a single, standard DSM diagram is too limited for that. Therefore we look at two extended DSM-based visualisation concepts. First, we consider the possibility of using a *linked series of DSMs*, where each DSM represents a single hierarchical level. While this would be a major improvement over a standard DSM diagram, it still does not fully satisfy requirement 1. Because of that, we propose a second visualisation approach which combines multiple hierarchical levels into a single *tree-based DSM diagram*. The second approach fully complies to requirement 1 and we have implemented it in our prototype.

Linked Series of DSMs

The concept of a linked series of DSMs builds further upon a concept we introduced in Chapter 3 (3.4.2) in connection with our spreadsheet generation tool. There, the idea was to use a couple of two DSMs, a "module-level" one and a "parameter-level" one, to simultaneously document dependencies on two hierarchical levels. Both diagrams could then be linked by letting the module-level DSM summarise the dependency values of the parameter-level DSM, either by summing or maximising them, per ordered pair of modules.

Here, we propose to extend this couple of DSMs to an unlimited series of DSMs, each representing a different hierarchical level of the studied system[5]. The DSMs in the series should be linked by summarising lower level dependencies through summing[6].

Equation 5.1 formally defines how the DSMs in such a hierarchical series are to be linked. We number the DSMs in the series so that the one that corresponds to the lowest hierarchical level is labelled with the number 1, and those representing higher levels are labelled with increasing numbers. Then, a dependency value from DSM^H accounts for all *direct* dependencies – those occurring strictly on the H-th level – between an ordered pair of modules situated on the H-th hierarchical level, and unless $H = 1$, also for all lower-level dependencies – which are summarised from $DSM^{(H-1)}$ by summing – between submodules of the ordered pair.

$$d^H_{i,j} = \begin{cases} \mathcal{D}^H_{i,j} + \sum\limits_{\forall p \in i,\ \forall q \in j} d^{(H-1)}_{p,q} & \text{for } H > 1 \\ \mathcal{D}^H_{i,j} & \text{for } H = 1 \end{cases} \tag{5.1}$$

In the equation:

- $d^X_{a,b}$ is a dependency value (from DSM^X), which represents the dependencies from a module a to another module b, both situated on the X-th level in the hierarchy;

[5] We hereby slightly alter the meaning, as defined in Chapter 3 (3.4.2), of modules containing (design) parameters, because modules now contain other modules *as their parameters* (or *submodules*).

[6] Summarising by maximising makes no sense here because we do not want to ignore any lower level dependencies on higher hierarchical levels.

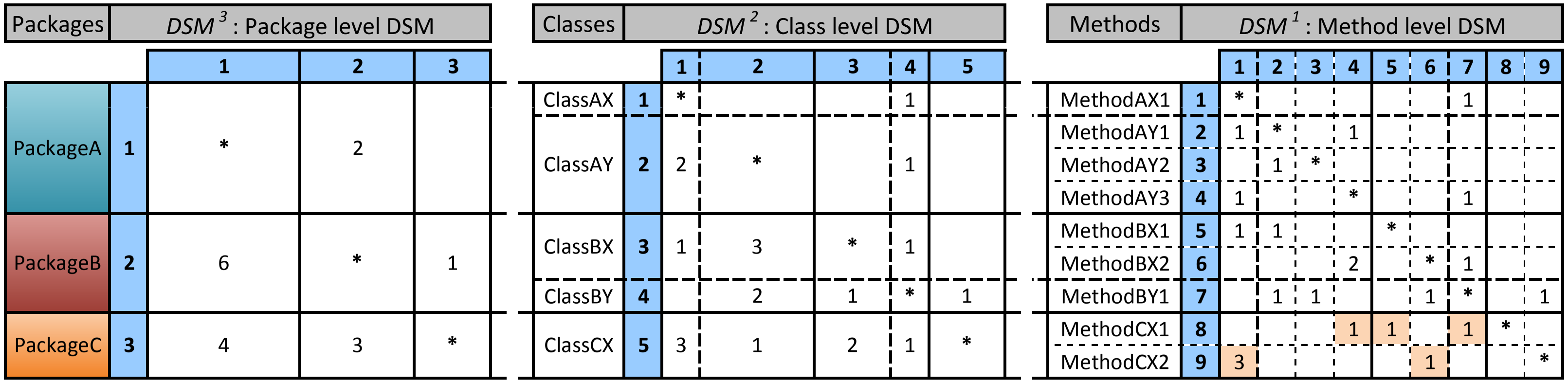
- $\mathcal{D}^X_{a,b}$ is a value that represents the *direct* dependencies, occurring strictly on the X-th hierarchical level, from a module a to another module b, both situated on the X-th level in the hierarchy.

For example, in the context of object-orientation a series of three DSMs could cover the levels of methods (DSM^1), classes (DSM^2) and packages (DSM^3). In that case, $\mathcal{D}^1_{m_1,m_2}$ would represent direct, *method-level* dependencies from a method m_1 to a method m_2 (e.g.: method calls), while $\mathcal{D}^2_{c_1,c_2}$ would represent direct, *class-level* dependencies from a class c_1 to a class c_2 (e.g.: inheritance).

On the next page, figure 5.1 shows such a method-class-package DSM series for a hypothetical piece of object-oriented software. We only included direct dependencies on the method-level (i.e.: $\forall p_1, p_2 : \mathcal{D}^3_{p_1,p_2} = 0$ and $\forall c_1, c_2 : \mathcal{D}^2_{c_1,c_2} = 0$), to emphasise how higher-level DSMs summarise the dependencies on lower levels (i.e.: DSM^2 sums method-level dependencies per ordered pair of classes and DSM^3 does so per ordered pair of packages).

Also on the next page, figure 5.2 displays an UML class diagram [103] which clarifies the architecture of the software system represented by the DSM series in figure 5.1.

For simplicity the UML diagram in figure 5.2 only displays dependencies originating from PackageC. We now use those dependencies to illustrate the summarising process in the DSM series shown in figure 5.1. We assume that all method-level dependencies are caused by method calls, which are given an individual weight of 1. Looking at DSM^1, we see that the dependencies originating from PackageC are caused by 7 method calls to 5 different methods (one is called 3 times from the same method). Next, in DSM^2 we see that the 7 method calls target methods in 4 classes (respectively 3, 1, 2 and 1 times). Finally, in DSM^3 we see that the method calls from PackageC target methods in classes of both PackageA and PackageB (respectively 4 and 3 times).



Packages		DSM^3 : Package level DSM		
		1	2	3
PackageA	1	*	2	
PackageB	2	6	*	1
PackageC	3	4	3	*

Classes		DSM^2 : Class level DSM				
		1	2	3	4	5
ClassAX	1	*			1	
ClassAY	2	2	*		1	
ClassBX	3	1	3	*	1	
ClassBY	4		2	1	*	1
ClassCX	5	3	1	2	1	*

Methods		DSM^1 : Method level DSM								
		1	2	3	4	5	6	7	8	9
MethodAX1	1	*						1		
MethodAY1	2	1	*		1					
MethodAY2	3		1	*						
MethodAY3	4	1			*			1		
MethodBX1	5	1	1			*				
MethodBX2	6				2		*	1		
MethodBY1	7		1	1			1	*		1
MethodCX1	8				1	1		1	*	
MethodCX2	9	3					1			*

Figure 5.1: Series of linked DSMs for the packages (DSM^3), classes (DSM^2) and methods (DSM^1) of a hypothetical piece of object-oriented software; for clarity only method-level dependencies are included

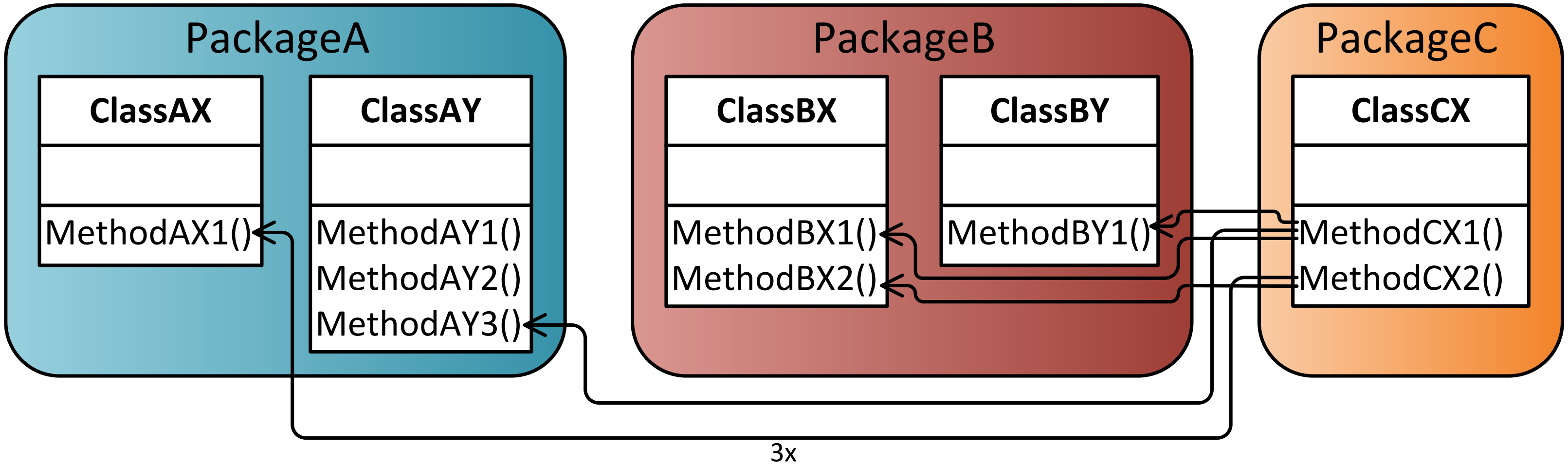
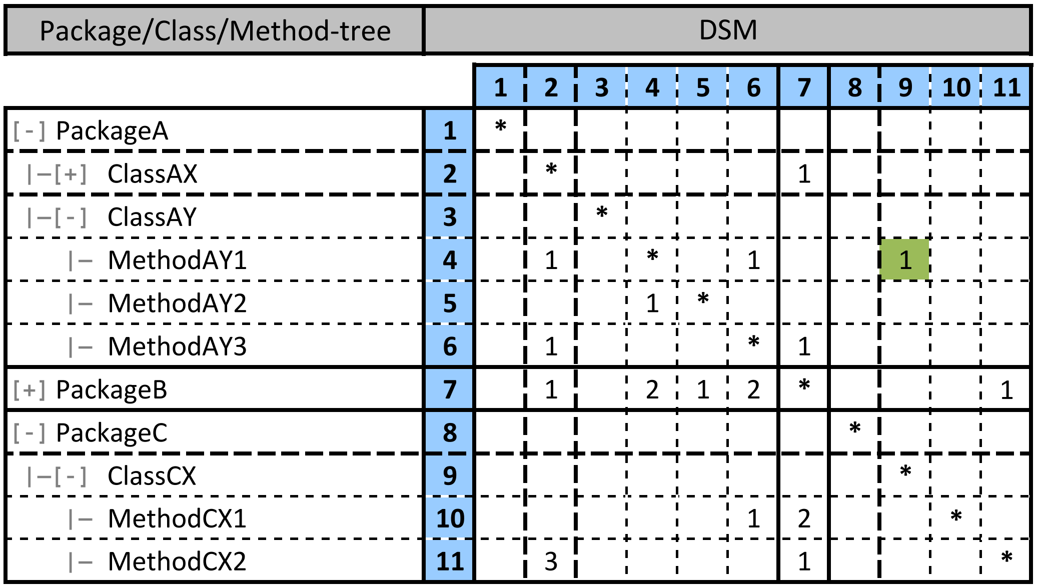


Figure 5.2: UML class diagram for the system from figure 5.1; for simplicity the diagram only displays the dependencies that originate from PackageC (which are highlighted in DSM_3)

Tree-based DSM diagram

A series of linked DSMs is a powerful way to display the hierarchical structure of object-oriented software. However, there are some issues we should deal with. For one thing, we expect that it might be difficult to integrate this concept in a convenient user interface. But perhaps more importantly, requirement 1 is not fully met because a series of DSMs, each of which represents a specific hierarchical level, fails to represent dependencies that intersect hierarchical levels. For instance, if MethodAY1 were to reference ClassCX by its name (e.g.: as the type of a variable), none of the DSMs in figure 5.1 would be able to express that dependency because methods and classes are never confronted in the same DSM.

To deal with both issues, we propose to integrate individual DSMs from a linked series into a single, *tree-based* DSM. In such a diagram, the module names column is be replaced by a tree-based view of the hierarchy of the system. Figure 5.3 illustrates what the result of applying this approach to the system from figures 5.1 and 5.2 could look like.



Package/Class/Method-tree	DSM											
		1	2	3	4	5	6	7	8	9	10	11
[-] PackageA	1	*										
|−[+] ClassAX	2		*					1				
|−[-] ClassAY	3			*								
|− MethodAY1	4		1		*		1			1		
|− MethodAY2	5				1	*						
|− MethodAY3	6		1				*	1				
[+] PackageB	7		1		2	1	2	*				1
[-] PackageC	8								*			
|−[-] ClassCX	9									*		
|− MethodCX1	10						1	2			*	
|− MethodCX2	11		3					1				*

Figure 5.3: Integration of multiple hierarchical levels in a tree-based DSM diagram

The tree-based DSM in figure 5.3 visualises all three hierarchical levels of the system, but it *abstracts away* the details of some, user-selected, modules (i.e.: ClassAX and PackageB) by showing them in a "collapsed" state – meaning that any lower-level modules they may contain are hidden. The other modules are shown in an "expanded" state – meaning that any lower-level modules they may contain are shown elsewhere in the DSM. The collapsed or expanded state of a module is respectively indicated by a + sign or a – sign in front of its name.

This approach does support representing dependencies which intersect hierarchical levels – as demonstrated by the highlighted dependency from MethodAY1 to ClassCX in the diagram in figure 5.3 – and thus fully complies with requirement 1.

Furthermore, a tree-based DSM diagram is ideal to be integrated in an interactive graphical user interface (GUI). The idea is to dynamically repaint the displayed DSM diagram when a user collapses of expands a module in an on-screen hierarchy tree.

Collapsing and expanding modules is respectively equivalent to zooming in and out on the visualisation and allows us to meet requirements 2 and 3. It facilitates hierarchical navigation through the system – for instance, by recursively expanding modules we can "drill down" to the source of a particular dependency. Lateral navigation can be achieved by selectively collapsing and/or expanding modules – for example, by collapsing those modules we do not and expanding those do care about, we can focus on what is important at a particular moment and avoid overloaded visualisations.

We should note that the choice to visualise software structures using tree-based DSMs was partially inspired by Lattix LDM, one of the applications we discuss as related work in section 5.5.

In summary, the tree-based DSM diagram resolves the issues of the linked series of DSMs and is therefore the preferred visualisation approach. Consequently, this is what we have implemented in the prototype we discuss in section 5.4.

To conclude, we present a formal definition, by means of equations 5.2, 5.3 and 5.4, of the dependency values in a tree-based DSM.

$$t_{i,j} \;=\; \mathcal{D}_{i,j} \;+\; h_{i,j} \tag{5.2}$$

$$h_{i,j} \;=\; \begin{cases} \sum\limits_{\forall p \in i} s_{p,j} \;+\; \sum\limits_{\forall q \in j} s_{i,q} \;+\; \sum\limits_{\forall p \in i,\ \forall q \in j} s_{p,q} & \text{when } i \text{ and } j \text{ are in collapsed state} \\ \sum\limits_{\forall p \in i} s_{p,j} & \text{when only } i \text{ is in collapsed state} \\ \sum\limits_{\forall q \in j} s_{i,q} & \text{when only } j \text{ is in collapsed state} \\ 0 & \text{when } i \text{ nor } j \text{ is in collapsed state} \end{cases} \tag{5.3}$$

$$s_{x,y} = \mathcal{D}_{x,y} + \begin{cases} \sum\limits_{\forall a \in x} s_{a,y} + \sum\limits_{\forall b \in y} s_{x,b} + \sum\limits_{\forall a \in x,\ \forall b \in y} s_{a,b} & \text{when } x \neq \emptyset \text{ and } y \neq \emptyset \\ \sum\limits_{\forall a \in x} s_{a,y} & \text{when } x \neq \emptyset \text{ and } y = \emptyset \\ \sum\limits_{\forall b \in y} s_{x,b} & \text{when } x = \emptyset \text{ and } y \neq \emptyset \\ 0 & \text{when } x = \emptyset \text{ and } y = \emptyset \end{cases} \tag{5.4}$$

In the equations:

- $t_{i,j}$ is a dependency value between a module i and another module j as it is displayed in a tree-based DSM; where i and j can be situated on different hierarchical levels of the represented system;

- $h_{i,j}$ represents the portion of $t_{i,j}$ that accounts for dependencies to and/or from modules which are hidden because of the *collapsedness* of i, j or both:

  - if both i and j are collapsed $h_{i,j}$ is taken as the recursive sum of the dependencies from all submodules of i to j, those from i to all submodules of j and those from all submodules of i to all submodules of j,
  - if only i is collapsed $h_{i,j}$ is taken as the recursive sum of the dependencies from all submodules of i to j,
  - if only j is collapsed $h_{i,j}$ is taken as the recursive sum of the dependencies from i to all submodules of j,
  - if i nor j is collapsed then all their submodules are visible elsewhere in the DSM, so $h_{i,j} = 0$;

- The value $\mathcal{D}_{x,y}$ represents all *direct* dependencies from a module x to a module y (and not from/to their submodules); where x and y can be situated on different hierarchical levels;

- $s_{x,y}$ recursively sums all dependencies from a module x, and its submodules, to another module y, and its submodules; where x and y can be situated on different hierarchical levels.

Furthermore:

- A module m, situated at a H-th hierarchical level, is empty ($m = \emptyset$) when it contains no submodules; possibly because it is situated on the lowest hierarchical level ($H = 1$);

- Modules are considered collapsed when they are not expanded and vice versa;

- Empty modules are considered to be expanded.

5.3.2.2 Source Code Browser

A source-code browser is a tool that presents the source code of a software system in a way that facilitates navigation (browsing), exploration and analysis of the implementation of the system. Usually the browser processes the code to generate some abstract representation based on a type of diagrams.

We believe that this is the ideal format to meet the requirements we specified earlier. We thus propose to create a source-code browser that uses the tree-based DSM diagram we discussed above, as a view on the code. Hence, the application should provide an on-screen DSM-based visualisation of the implementation of the studied system, where the elements of the DSM – modules and dependencies – are abstract representations of underlying source code constructs.

The browser should be integrated into a development environment to support users in their day-to-day activities. Unlike stand-alone diagram generation tools, IDE-integrated code browsers are "online" all the time. This means that the provided representation can stay synchronised with the source code "reality", through automatic regeneration upon changes in the source code. These on-the-fly updates allow the visualisation to *co-evolve* [17, 76] with the actual implementation of the studied software.

We believe that co-evolving DSM diagrams could be an excellent tool to support dependency management in OOSD, because they create a direct, causal link between changes in the source code and resulting shifts in the explicitly visualised dependency patterns.

Requirement 3 demands that the source code processing should not result in a static, purely graphical, visualisation of the system. Instead, the visualisation should enable interactive navigation through the implementation of the system and users should be able to easily move back and forth between the visualisation and the source code. To make it dynamical, we propose to back the on-screen DSM view with a model[7] in which the individual elements that constitute the DSM remain semantically linked to the specific source code constructs they represent. This tight coupling could then support a number of interesting, interactive features in the user interface. For instance, context menus could be used to offer specific functionality for a user-selected element in the DSM (e.g.: expand or collapse a module, show underlying source code, track all dependencies from or to a module, inspect the target of a dependency, etc.).

5.3.2.3 Metaprogramming-based Analysis

Requirement 4 stipulates that the tool should offer a set of analysis features that can help developers to find and evaluate modularisation opportunities in the implementation of the

[7] As in the Model-View-Controlled design pattern [58, 59, 60].

studied software system. These opportunities represent changes in the implementation, that potentially lower the degree of dependency between some of its components and thus improve its modularity properties. Specifically, users may want to know if and where local architectural restructurings can be introduced, by means of common object-oriented strategies, to achieve such modularity improvements.

The kind of modularisation opportunities and the conditions that govern their applicability vary greatly with the context. Hence, the analysis features should be highly customisable. Therefore, we propose to equip the source-code browser with metaprogramming facilities. Using these facilities, the users themselves can write ad-hoc metaprograms to inspect (parts of) specific systems in search of specific information, with regard to modularisation opportunities or other purposes.

To this end, the tool should expose a framework of abstractions as an API which metaprograms can employ to reflect on the source code from a higher level of abstraction. We believe that the model which backs the DSM visualisation (see above) could be reused for this purpose. Primarily because it describes an abstract representation in terms of modules and dependencies, which is ideal to let metaprograms reason about the modularity and dependency properties of the system, in search of modularisation opportunities. Secondly, because metaprograms can use the elements of the DSM model as a middleman to access underlying implementation constructs (i.e.: for investigations that can only be carried out on the actual source code).

Although users should have the freedom to write metaprograms for all sorts of purposes, the main objective is to enable them to write programs that automate the process of gathering knowledge with regard to modularisation opportunities. The kind of information that is collected may vary depending on the context and the level of sophistication of the program: ranging from plain statistical data amassed by simple metaprograms, to concrete, directed modularisation suggestions supplied by more advanced metaprograms.

While the task of writing metaprograms for specific purposes be only be left to the users, the tool should provide some generic examples and a collection of essential building blocks. Futhermore, the user interface should provide hooks to run metaprograms against specific parts, selected from the on-screen DSM visualisation, of the studied system.

We should note that these metaprogramming-based analysis features motivate why we chose the format of an IDE-integrated code browser in the first place – instead of a standalone ("offline") tool. The reason is that the format enables us to provide metaprograms with an abstract representation that remains connected to the underlying implementation constructs, which greatly simplifies reasoning about the system. Furthermore, we expect that a hierarchical representation in terms of modules and dependencies, as provided by the tree-based DSM model, is ideal to let metaprograms search for modularisation opportunities.

5.3.2.4 Summary

We expect that tree-based DSM-based visualisation, source-code browsers and metaprogramming facilities form a powerful combination for support tools for OOSD.

We believe that an implementation of the approach we present, could effectively assist developers in common tasks related to dependency management. These tasks range from passive information gathering (e.g.: tracking which types of dependencies occur how frequently in which locations), to active interventions (e.g.: improving modularity by minimisation, centralisation or even elimination of dependencies). A tool that implements our approach should be able to support developers in these tasks, both passively (e.g.: by providing users with a practical way to navigate through the system to manually track dependencies or search for modularisation opportunities) and actively (e.g.: through metaprograms that automate information gathering or even supply modularisation suggestions).

While to primary goal is to support dependency management, we think such a tool could also help developers to explore and familiarise themselves with large amounts of unknown source code. For instance when they have to maintain, reengineer or replace legacy code.

To validate these assumptions we have developed a prototype tool, which we discuss in the next section, and a number of usage scenarios, which are covered by Chapter 6.

5.4 DSMBrowser

In this section we present *DSMBrowser*[8], a prototype of an extendable DSM-based source-code browser with analysis capabilities based on metaprogramming. To meet the specified requirements this implementation closely follows the approach we described above.

In what follows we introduce the context, the most prominent features and the important architectural details of DSMBrowser.

5.4.1 Context

We implemented DSMBrowser in the Smalltalk programming language and it is intended to support OOSD in Smalltalk environments. The choice for Smalltalk as the implementation language was inspired by its extensive support for metaprogramming and reflection through its meta-object protocol (MOP).

DSMBrowser was developed using Cincom VisualWorks [105], an IDE for Smalltalk, and it

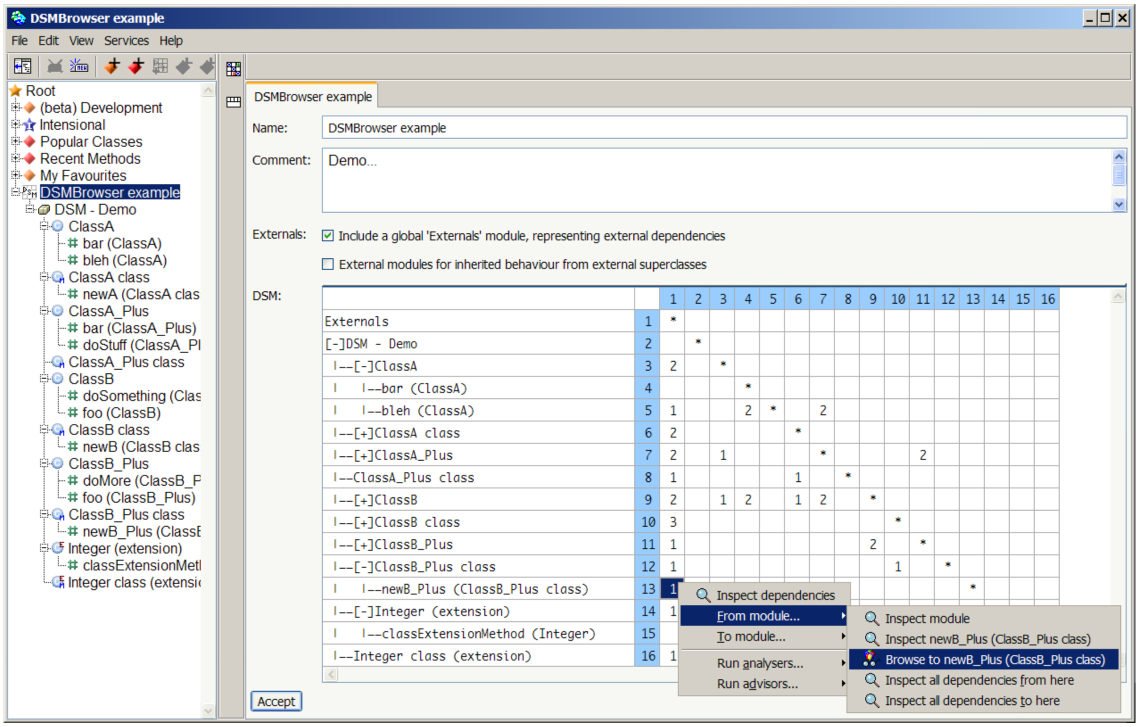
[8] DSMBrowser is available at the Cincom Public Store Repository (http://www.cincomsmalltalk.com/CincomSmalltalkWiki/PostgreSQL+Access+Page).

integrates with that same environment to support the development of other software. More specifically, DSMBrowser is an extension of StarBrowser, a code classification and browsing tool for VisualWorks, developed by Roel Wuyts [77, 80].

In line with requirement 5, the DSMBrowser system was conceived as an open-ended framework, which can be extended and customised in a number of ways, which we mention below.

5.4.2 User Interface

DSMBrowser has a graphical user interface which is closely integrated into StarBrowser and VisualWorks. Figure 5.4 shows a screenshot.



Name: DSMBrowser example

Comment: Demo...

Externals: ☑ Include a global 'Externals' module, representing external dependencies

☐ External modules for inherited behaviour from external superclasses

DSM:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Externals	1	*															
[-]DSM - Demo	2		*														
|--[-]ClassA	3	2		*													
| |--bar (ClassA)	4				*												
| |--bleh (ClassA)	5	1			2	*		2									
|--[+]ClassA class	6	2					*										
|--[+]ClassA_Plus	7	2		1				*				2					
|--ClassA_Plus class	8	1					1		*								
|--[+]ClassB	9	2		1	2		1	2		*							
|--[+]ClassB class	10	3									*						
|--[+]ClassB_Plus	11	1								2		*					
|--[-]ClassB_Plus class	12	1									1		*				
| |--newB_Plus (ClassB_Plus class)	13	1												*			
|--[-]Integer (extension)	14	1															
| |--classExtensionMethod (Integer)	15																
|--Integer class (extension)	16	1															

Figure 5.4: DSMBrowser integrated into the StarBrowser interface

The tree view widget on the left-hand side of the window is part of the StarBrowser interface and lists different kinds of classifications and their contents. To use DSMBrowser a user must add a special *DSMView* classification to the classifications list, by clicking a button on the toolbar. The DSMView can then be populated by dragging and dropping entities of Smalltalk code from *Refactoring Browser* [61, 11] windows[9] or from other classifications in the StarBrowser.

[9] The current Visual Works System Browser is an implementation of the Refactoring Browser of Roberts et al. [61, 11].

By using the classification model [80], DSMBrowser gives developers the ability to select arbitrary entities of Smalltalk code as the modules to confront in a DSM. This helps them to focus on the implementation parts of their choice. It is even possible to group modules across multiple hierarchies. For instance, one can easily study the dependencies between a single class and a whole package by simply adding the class (without its own surrounding package) and the package to a DSMView classification.

Once the DSMView has been populated a tree-based DSM visualisation will show up in the right-hand side of the window. This visualisation offers all the interactivity features we described earlier. Modules can be dynamically collapsed and/or expanded to support hierarchical and lateral navigation. Furthermore, context menus – like the one on the screenshot in figure 5.4 – offer specific functionality with regard to individual modules or dependencies. Examples include opening a Refactoring Browser to look at source code, opening a *Trippy* inspector [106] to inspect an object or running a metaprogram against a set of dependencies.

5.4.3 DSM Model

Before DSMBrowser can visualise the system formed by the selected source code entities, it must process the code to build an in-memory representation of the modular structure of the system and all occurring dependencies. This *DSM model* will then back the on-screen visualisation and can be employed by metaprograms as an API to reason about the system.

Figure 5.5, on the next page, shows a simplified UML class diagram [103] of the architecture of the DSM model. To reflect the hierarchical structure of studied code entities, the design uses the Composite pattern [22]. In a nutshell the architecture is set up as follows:

- all components in the hierarchy are represented by instances of the Module class;
- the diagram as a whole is an instance of the DSM class, a subclass of Module;
- all Module instances are composites that can contain other Module instances as their *parameters* (or *submodules*);
- every Module instance holds a reference to its *parent* Module and to its *subject*:
  - the subject is the source code entity that the Module instance represents,
  - the subject of a DSM object is a DSMView classification;
- a Dependency instance represents an implementation-level dependency, from the subject of its *from-* (or *source*) Module instance, to the subject of its *to-* (or *target*) Module instance:
  - it contains references to those Module instances, as well as a weight value (which defaults to 1),

  - the Dependency class represents generic dependencies, subclasses can be defined to represent different types of dependencies (possibly with adjusted weights);

- A DSM instance contains a reference to a DependencyDictionary which indexes Dependency objects by their source and their target Module.

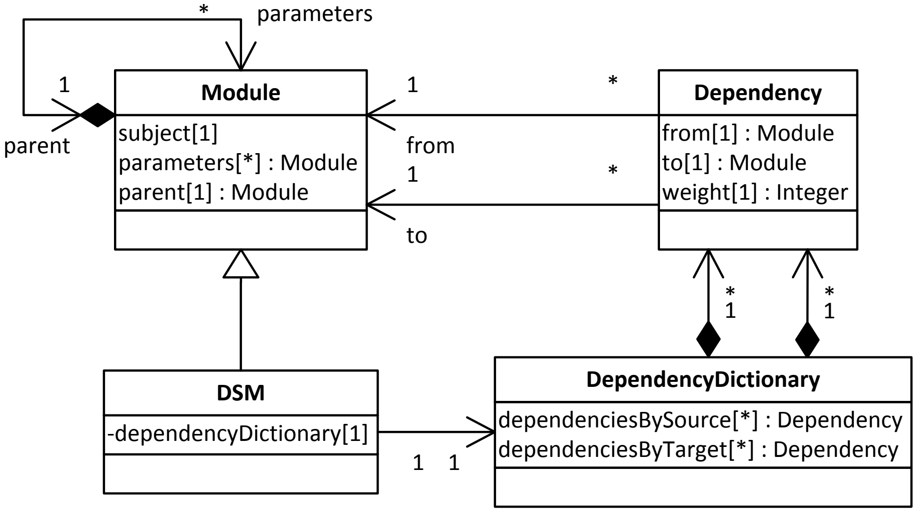


Figure 5.5: UML class diagram [103] for the DSM model architecture (simplified)

DSMBrowser extends the service architecture of StarBrowser, which implements the Visitor design pattern [22]. To build a DSM model for selected entities of code, DSMBrowser uses two important services.

First, a service that determines the *children* of the subject of a module (e.g.: the methods of a class), is used to construct a tree representation. This tree is a parallel representation of the hierarchical structure of the code entities. Building the tree is a recursive process: for every module the children of its subject are determined and these children then become the subjects of the submodules of the module. The process starts with the subject of the DSM, which is a DSMView classification whose children are the *root* code entities (the ones the user dragged and dropped into the classification). The process stops when the bottom of the hierarchy is reached.

Once the tree is completed, DSMBrowser uses a service which determines the dependencies originating from each module in the tree. The dependencies are stored in the Dependency-Dictionary of the DSM. Because the target of some dependencies might not be present in the classification (e.g.: a class might inherit from a class that is contained in a package that was not added to the DSMView), users can choose to add a “virtual” module that acts as a placeholder for external dependency targets[10].

[10] This feature was inspired by the work of Sullivan et al. [72], which is discussed in Chapter 4 (section 4.3).

The DSM Module design is conceptually independent of the Smalltalk language because all language specific behaviour is implemented in the services. Therefore, the design could be straightforwardly reimplemented in another DSM-browser to target another object-oriented programming language. Furthermore, future versions of DSMBrowser could be extended with support for other languages by using intermediary libraries.

5.4.4 Smalltalk Language Support

In DSMBrowser, pretty much any entity of Smalltalk source code can act as a module in the DSM diagram. We support bundles, packages, classes, namespaces and methods. Moreover, we differentiate meta classes from regular classes, class extensions from class definitions and namespace extensions from namespaces definitions.

DSMBrowser can express various kinds of relationships among source code entities as dependencies. Currently we support message sends, class name references, inheritance relationships, class extensions and namespace relationships. New kinds of dependencies can be added by defining new subclasses of Dependency and extending the dependency service. A possible candidate is a dependency for overriding relationships (i.e.: from an overriding method to the overridden method), which would be trivial to add.

The dependency service tracks the targets of dependencies using different techniques depending on the type of dependency. Target tracking is exact for all types expect message sends. Because Smalltalk is a dynamically typed language it is not easy to determine the type of the object a message is being sent to. Currently, we use a fairy naïve heuristic that creates dependencies to all classes that implement a method with the same name as the message that is being sent. Obviously the performance of this heuristic is rather poor, especially when common Smalltalk methods (e.g.: `new`, `initialize`, `release`, etc.) are concerned. However, experiments have shown that this does not at all render the tool useless. This problem is entirely due to the Smalltalk typing system and would not occur in a similar code browser for a statically typed language. Moreover, future versions of the DSMBrowser could use more sophisticated heuristics, for instance based on the techniques used in RoelTyper [78], a tool that infers the types of instance variables in Smalltalk using advanced heuristics.

5.4.5 Analysis Features

The choice for Smalltalk, as the implementation language of DSMBrowser, made it easy to equip the tool with the metaprogramming-based analysis features we specified earlier (see 5.3.2.3). Smalltalk's metaprogramming support and the open VisualWorks environment enable users to implement analysing metaprograms in plain Smalltalk. However, DSMBrowser does provide a number of facilities that considerably simplify that task.

As mentioned earlier, the DSM model was designed to act both as a backend for the DSM visualisation and as an API to be targeted by user-written metaprograms, primarily with regard to finding modularisation opportunities. Therefore, much of the model's classes implement functionality that is specifically intended to support such programs. For instance, the DependencyDictionary class provides a whole range of querying methods that can be used mine for specific dependencies or the modules they connect.

Additionally, DSMBrowser contains an extendable dependency filtering infrastructure that can be used to filter and sort dependencies. Besides generic filtering classes a collection of ready-to-use filters is available, including filters that throw out dependencies that:

- are not of a specific type (e.g.: only keep inheritance dependencies);
- are internal to a specific module – practical to hide dependencies among the classes and methods of a package when we only care about dependencies to other packages;
- result from the sending of common Smalltalk messages – useful given the accuracy of the current message target tracking heuristic.

Furthermore, DSMBrowser includes a small dependency analysis framework that provides essential building blocks for analysing metaprograms. The framework defines generic dependency visitors [22] and generic classes that implement the elementary behaviour for two basic types of metaprograms. The first class, DependencyAnalyser, can be subclassed to create metaprograms that strictly collect (statistical) information. The second one, DependencyAdvisor, acts as a superclass for metaprograms that actually evaluate the feasibility of certain modularisation opportunities and produce indicative suggestions or even specific advice, with regard to modularity-enhancing architectural restructurings. The GUI logic dynamically adds entries for subclasses of both metaprogram classes to the context menu of the DSM visualisation. That way, new user-defined analysis programs become immediately available within the DSMBrowser interface to run against sets of dependencies.

To demonstrate the analysis capabilities of DSMBrowser, we have developed a few metaprograms that assist users with finding and evaluating modularisation opportunities and serve as examples for user-written programs. While they are intended as a proof of concept these metaprograms are usable in real-world situations, as we show in Chapter 6.

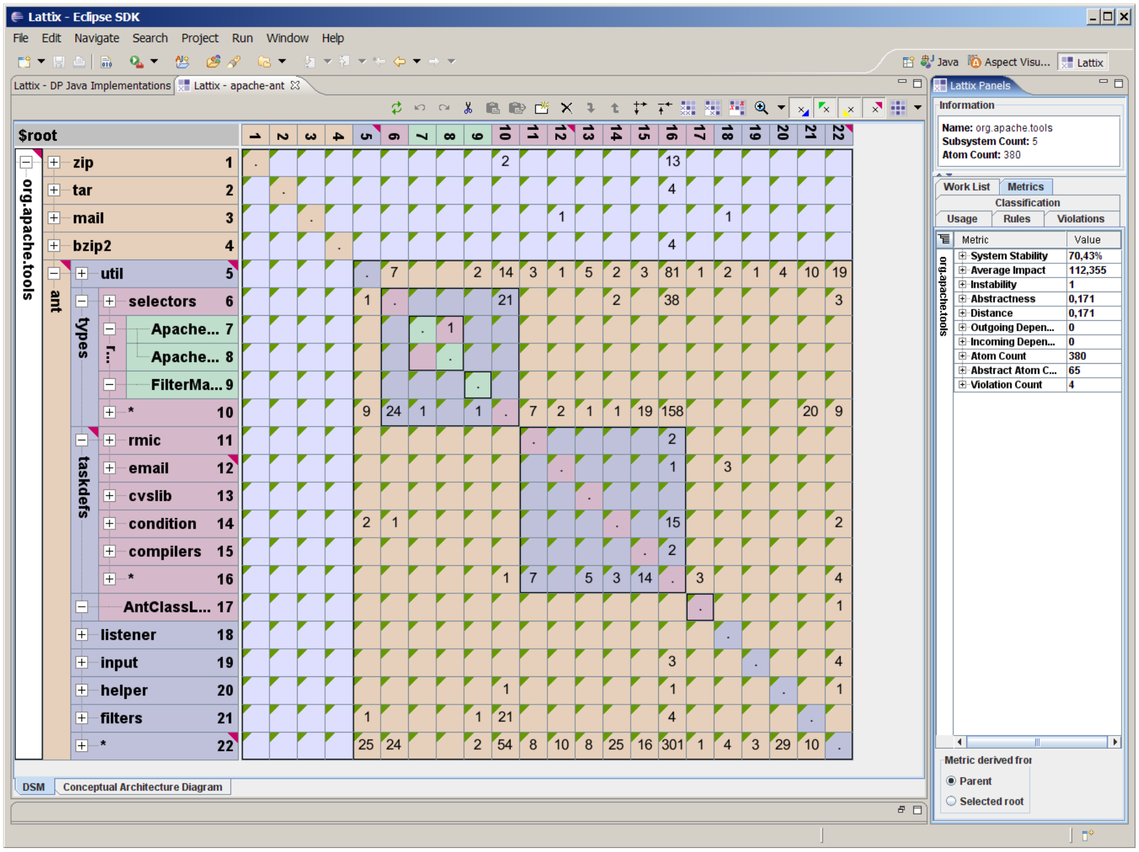
5.5 Related Work

Although the application of DSMs in support of software development is a fairly recent phenomenon, other parties have conducted related research and created similar tools. We discuss three such tools, Lattix LDM and NDepend, which are the most mature examples, and an experimental program called DeMatrix.

5.5.1 Lattix LDM

Lattix, Inc. was the first company to release a commercial support tool for software development which applies DSMs as abstract representations of software implementations. The product is called Lattix LDM [95, 26] and is primarily promoted as a tool for analysing and managing large-scale software development projects. Lattix LDM and its underlying methodology have been demonstrated in talks at conferences [63, 64] and in a number of articles [41, 62, 65]. We have experimented with a trial version of Lattix LDM[11].

Lattix LDM reverse engineers Java, C/C++ and .NET code to DSM diagrams and comes as a stand-alone application (for Windows or Linux) and as a plug-in for the Eclipse development environment [92]. Figure 5.6 shows a screenshot of the Eclipse plug-in displaying a DSM visualisation of the Apache Ant [87] source code. We created this DSM by reproducing the procedure explained in [41].



$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
zip	1	.									2						13						
tar	2		.														4						
mail	3			.									1						1				
bzip2	4				.												4						
util	5					.	7			2	14	3	1	5	2	3	81	1	2	1	4	10	19
selectors	6					1	.				21				2		38						3
Apache...	7							.	1														
Apache...	8								.														
FilterMa...	9									.													
*	10					9	24	1		1	.	7	2	1	1	19	158					20	9
rmic	11											.					2						
email	12												.				1		3				
cvslib	13													.									
condition	14					2	1								.		15						2
compilers	15															.	2						
*	16										1	7		5	3	14	.	3					4
AntClassL...	17																	.					1
listener	18																		.				
input	19																3			.			4
helper	20										1						1				.		1
filters	21					1				1	21						4					.	
*	22					25	24			2	54	8	10	8	25	16	301	1	4	3	29	10	.

Name: org.apache.tools
Subsystem Count: 5
Atom Count: 380

Metric	Value
System Stability	70,43%
Average Impact	112,355
Instability	1
Abstractness	0,171
Distance	0,171
Outgoing Depen...	0
Incoming Depen...	0
Atom Count	380
Abstract Atom C...	65
Violation Count	4

Figure 5.6: Lattix LDM [95] Eclipse plug-in showing a DSM for Apache Ant [87]

The user interface of Lattix LDM uses a tree-based DSM visualisation, which aggregates classes per package level and allows packages to be collapsed and expanded. While this was

[11] Version 2.7.5; while version 3.1 was recently released, licensing constraints do not allow us to use it.

a source of inspiration for DSMBrowser, Lattix does not offer the same level of detail, as it does not document dependencies below the level of classes.

Lattix computes package-level dependencies by summing class-level dependencies. Class-level dependency values (which are referred to as dependency strengths) are numerical and can be configured to be *knowledge-based* on *usage-based*. The available documentation lacks a formal explanation of both configurations, but as far as we can tell the knowledge-based configuration only takes into account which classes "know" one another, while the usage-based configuration expresses the degree to which classes use each other's functionality. Knowledge-based dependency strengths seem to be limited to a scale from 0 to 2. The usage-based configuration results in a much wider range of dependency strengths and clearly provides a more detailed approximation of implementation level dependencies. However, due to the lack of method-level dependencies, even Lattix LDM's usage-based configuration fails to provide the same level of detail DSMBrowser offers.

Other notable features of Lattix LDM include a number of dependency filtering settings and the definition of design rules[12] to capture and enforce architectural intent. Software architects can create design rules with Lattix LDM to express the nature of dependencies between subsystems or classes. Dependencies that violate such design rules are then highlighted in the visualisation.

The classification model [80] used in DSMBrowser allows the user to confront any mix of arbitrary source code entities in a DSM. In comparison, Lattix LDM does not offer the same level of flexibility.

While it provides various statistics and metrics[13] to analyse software systems, Lattix lacks support for analysis through metaprogramming and does not provide suggestions with regard to modularisation opportunities.

5.5.2 NDepend

NDepend [99], a product of Smacchia.com s.a.r.l., is another commercial software development tool that applies DSM diagrams[14]. It is a dependency management tool that is intended to facilitate controlling the complexity, quality and evolution of source code. NDepend exclusively targets the .NET software development platform [98] and integrates with the Microsoft Visual Studio IDE [97].

The tool analyses source code and compiled .NET assemblies to generate a reports and

[12] Refer to Chapter 2 (2.3.3) for an explanation of the concept of design rules.

[13] These metrics do not include Net Option Value calculation.

[14] This information is solely based on texts on the NDepend website [99], as we have not experimented with NDepend ourselves.

interactive graphical visualisations, based on tree-based DSMs and other diagrams. It also includes over 60 predefined metrics to analyse different aspects of software implementations. Moreover, it provides metaprogramming facilities by means of an SQL-like query language called Code Query Language (CQL), which allows users to write queries against the code structure of .NET applications and which can be used to write custom metrics. This query-based approach differs considerably from the metaprogramming API of DSMBrowser. Which approach is superior in practice cannot be decided at present.

5.5.3 DeMatrix

DeMatrix [2] is a tool created by Sushil Bajracharya et al. at the University of California, Irvine. It was developed in connection with a larger research project that aims to create an infrastructure, named Sourcerer [3, 101], for large-scale analysis of open source code repositories. DeMatrix is a front-end for Sourcerer that visualises software using DSMs.

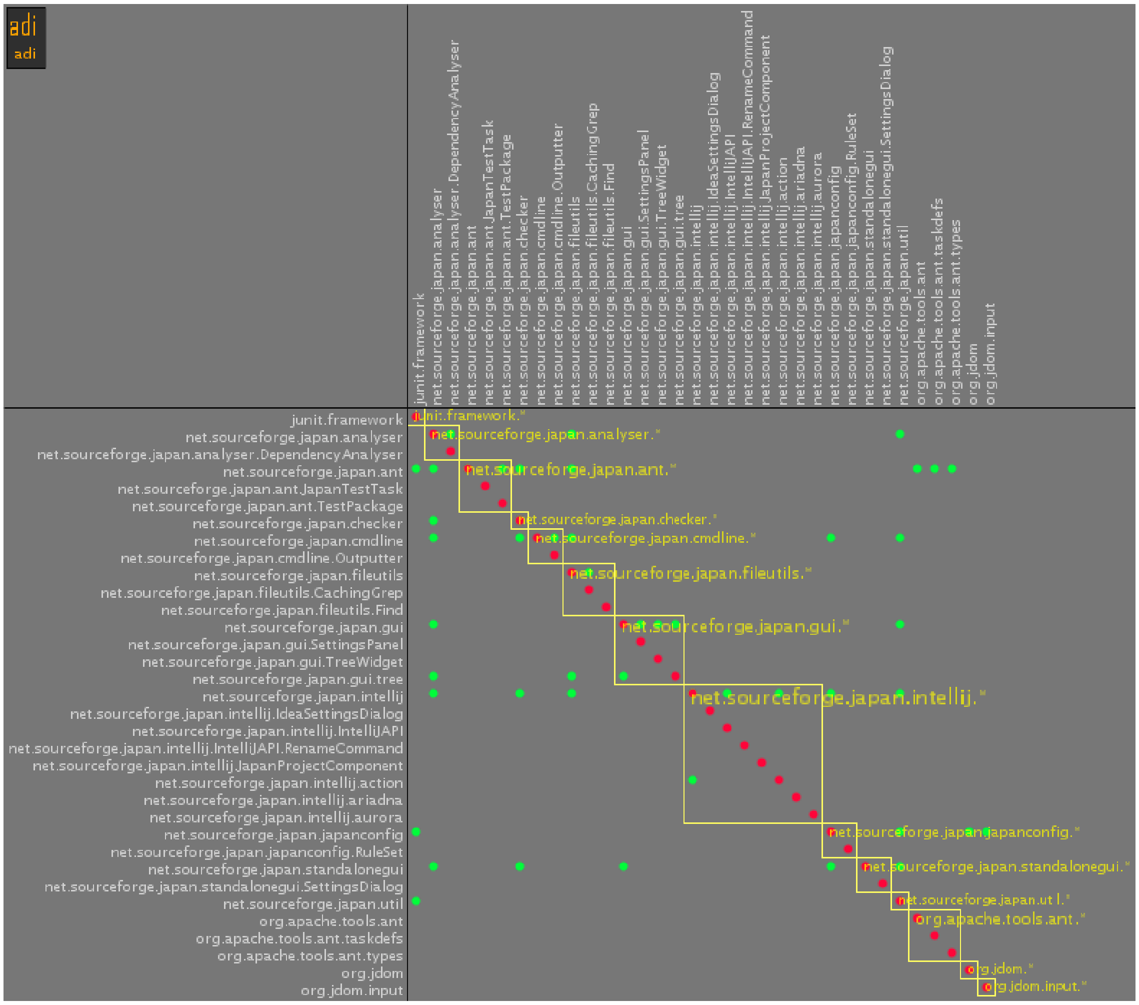

adi
adi
junit.framework
net.sourceforge.japan.analyser
net.sourceforge.japan.analyser.DependencyAnalyser
net.sourceforge.japan.ant
net.sourceforge.japan.ant.JapanTestTask
net.sourceforge.japan.ant.TestPackage
net.sourceforge.japan.checker
net.sourceforge.japan.cmdline
net.sourceforge.japan.cmdline.Outputter
net.sourceforge.japan.fileutils
net.sourceforge.japan.fileutils.CachingGrep
net.sourceforge.japan.fileutils.Find
net.sourceforge.japan.gui
net.sourceforge.japan.gui.SettingsPanel
net.sourceforge.japan.gui.TreeWidget
net.sourceforge.japan.gui.tree
net.sourceforge.japan.intellij
net.sourceforge.japan.intellij.IdeaSettingsDialog
net.sourceforge.japan.intellij.IntelliJAPI
net.sourceforge.japan.intellij.IntelliJAPI.RenameCommand
net.sourceforge.japan.intellij.JapanProjectComponent
net.sourceforge.japan.intellij.action
net.sourceforge.japan.intellij.ariadna
net.sourceforge.japan.intellij.aurora
net.sourceforge.japan.japanconfig
net.sourceforge.japan.japanconfig.RuleSet
net.sourceforge.japan.standalonegui
net.sourceforge.japan.standalonegui.SettingsDialog
net.sourceforge.japan.util
org.apache.tools.ant
org.apache.tools.ant.taskdefs
org.apache.tools.ant.types
org.jdom
org.jdom.input
junit.framework.*
net.sourceforge.japan.analyser.*
net.sourceforge.japan.ant.*
net.sourceforge.japan.checker.*
net.sourceforge.japan.cmdline.*
net.sourceforge.japan.fileutils.*
net.sourceforge.japan.gui.*
net.sourceforge.japan.intellij.*
net.sourceforge.japan.japanconfig.*
net.sourceforge.japan.standalonegui.*
net.sourceforge.japan.util.*
org.apache.tools.ant.*
org.jdom.*
org.jdom.input.*

Figure 5.7: DeMatrix applet [2] displaying a DSM for the source code of JAPAN [93]

Currently DeMatrix is only available as a Java applet embedded on a demonstration webpage [2]. The applet displays DSM visualisations of Java source code of various open source projects hosted at SourceForge [100]. The screenshot in figure 5.7, on the previous page, shows the DeMatrix applet visualising the code of the JAPAN [93] project.

DeMatrix provides a basic, fairly static visualisation based on (binary) DSMs and no analysis features. The design parameters, which are confronted in the DSM, correspond to Java classes. Modules, corresponding to packages, are indicated using yellow bordered boxes on the diagonal. The DSM is not tree-based so packages cannot be collapsed or expanded. The module level can be shifted to correspond to a higher or a lower package level, which respectively results in larger or smaller module boxes but does not otherwise alter the visualisation.

5.6 Conclusions

The main contributions of the work we presented here are the rationale for a novel DSM-based support tool for OOSD and DSMBrowser, a prototype implementation of such a tool.

As part of the rationale we presented five essential requirements for a dependency management tool for OOSD, following from observations about the discipline. Considering those requirements we concluded that the ideal solution would be to combine tree-based DSM visualisations and metaprogramming-based analysis facilities in an extendable, IDE-integrated source-code browser. While there are other DSM-based support tools for software development, this combination of features forms an innovative and powerful approach.

DSMBrowser is an experimental implementation of this approach, applied to the Smalltalk language. It seamlessly integrates with StarBrowser and the VisualWork environment and has excellent support for the modularity and dependency characteristics of Smalltalk and object-oriented programming languages in general. The tool provides practical facilities that enable users to write simple, ad-hoc metaprograms to analyse software systems and to find and evaluate modularisation opportunities which improve software design.

Despite its experimental status, the case studies we discuss in Chapter 6 demonstrate that its unique combination of features makes DSMBrowser highly useful in real-world situations.

DSMBrowser can also serve as a test bed for future research with regard to applications of DSMs in support of software development. In Chapter 7 (section 7.3) we formulate some directions for this future work.

Chapter 6

DSMBrowser Case Studies

This chapter discusses a number of case studies that demonstrate how software developers can use DSMBrowser, the DSM-based source code browser we presented in the previous chapter, in real-world situations and how they can benefit from that. The case studies focus on the exploration of unknown source code and on common dependency management tasks. The most prominent contribution we present here is formed by metaprograms that automate the task of finding and evaluating modularisation opportunities.

6.1 Introduction

In Chapter 5 we introduced a rationale for a novel support tool for Object-Oriented Software Developments (OOSD) in the form of an extendable DSM-based source code browser with analysis capabilities based on metaprogramming. We also presented a prototype implementation of such a tool – named *DSMBrowser* – and we made a number of claims and assumptions with regard to the utility of this or future similar tools.

To demonstrate the prototype and validate our claims and assumptions, this chapter presents a number of case studies. Each case study is situated in a realistic *context* related to OOSD and presents a *usage scenario* which shows how and why software developers can benefit from using DSMBrowser in that context. Everything is illustrated with concrete examples and the usage scenarios offer tutorial-style explanations of how we propose to approach those.

We start out with an exploration case study in section 6.2. While supporting the exploration of unknown source code is not the primary goal of DSMBrowser, the scenario we discuss here helps to explain how the basic visualisation and navigation features can be used. Next, in section 6.3 we consider case studies regarding dependency management. The first case study deals with gathering of dependency information and the next ones deal with searching and evaluation of modularisation opportunities through metaprogramming. We conclude this chapter in section 6.4.

We should note that all DSM diagrams shown by figures in this chapter are screenshots of the DSM visualisation provided by DSMBrowser, to which we added coloured frames to highlight the important elements.

6.2 Source Code Exploration

Context

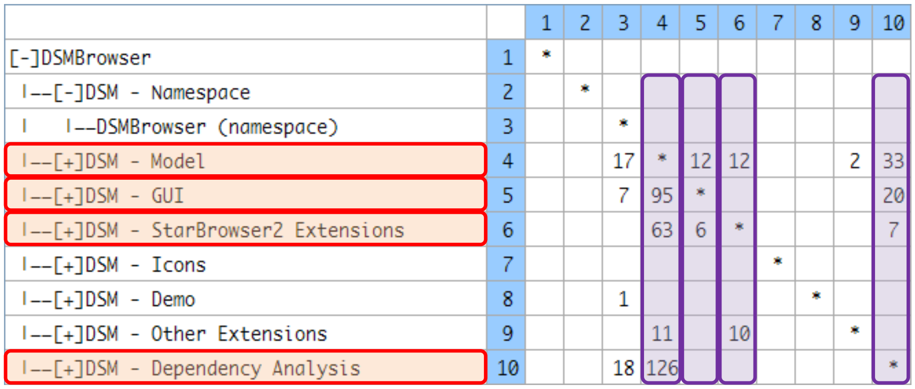
While the primary goal of DSMBrowser is to support dependency management, it can also help software developers to explore and familiarise themselves with large amounts of unknown source code.

There are many situations where developers have to explore unknown systems. For instance, when a new employee is hired to join an ongoing development project, he or she will have to become familiar with the system that is being built as quickly as possible. Furthermore, it is not uncommon that software developers are instructed to maintain, reengineer or replace legacy code that lacks documentation or which was written by people who already left the company.

To demonstrate how DSMBrowser can support code exploration and as a tutorial to the basic functionality of the tool, we explain how others could use it to familiarise themselves with the source code of DSMBrowser itself.

Usage Scenario

The tree-based DSM diagram in figure 6.1 shows an overview of all the packages in the DSMBrowser bundle[1]. The diagram was generated by creating a new DSMView classification in StarBrowser, to which the DSMBrowser bundle was added. This visualisation is an excellent starting point for exploration of the system.



		1	2	3	4	5	6	7	8	9	10
[-]DSMBrowser	1	*									
|--[-]DSM - Namespace	2		*								
| |--DSMBrowser (namespace)	3			*							
|--[+]DSM - Model	4			17	*	12	12			2	33
|--[+]DSM - GUI	5			7	95	*					20
|--[+]DSM - StarBrowser2 Extensions	6				63	6	*				7
|--[+]DSM - Icons	7							*			
|--[+]DSM - Demo	8			1					*		
|--[+]DSM - Other Extensions	9				11		10			*	
|--[+]DSM - Dependency Analysis	10			18	126						*

Figure 6.1: Exploring the implementation of DSMBrowser – Package overview

[1] Below we will explain why one of the packages is shown in expanded state.

Because it is not convenient to study all parts of the system simultaneously, the first step is to select which parts (packages) will be investigated first. The inter-package dependencies shown in the columns and rows of the DSM are helpful to make this choice.

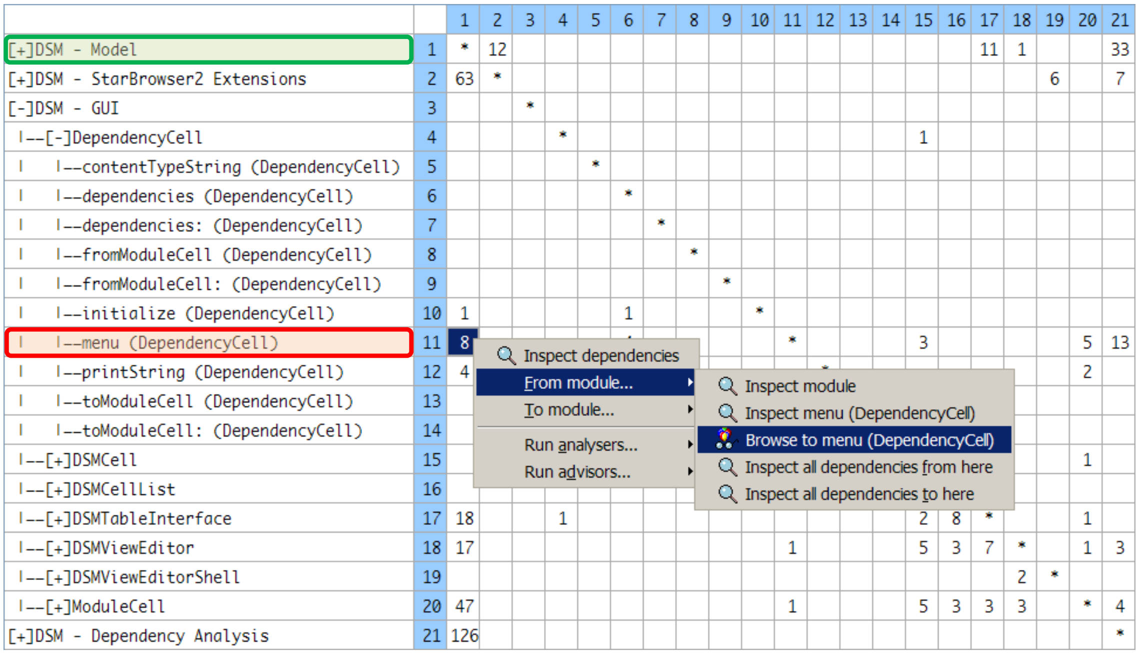
By looking at the columns, we see which packages are targeted the most by dependencies originating from other packages. Clearly some packages are more "popular" than others. At a glance, there are five popular packages: `Namespace`, `Model`, `GUI`, `StarBrowser2 Extensions` and `Dependency Analysis`. Assuming that the packages are named according to their contents or functionality we can suspect that the `Namespace` package only contains namespace definitions. By simply expanding the package – as we did on the figure – we can establish that this is indeed the case. We can now conclude that the high number of dependencies to this package is simply caused by the fact that every class in the system is defined within the `DSMBrowser` namespace[2]. Clearly the `Namepace` package is uninteresting from a functional point of view and does not require further investigation. Therefore, in the remainder of our scenario we only consider the other four popular packages.

By looking at the rows we see that the dependency interest is mutual, as the popular packages are also the ones which have the most dependencies to others. We can thus conclude that some packages are heavily dependent on one another, while others are relatively independent. Although the relative independence of a package does not guarantee that it is functionally unimportant, it does signify that we can study it in isolation from others. Consequently, it is probably wise to explore the set of four tightly connected packages first and to leave the independent ones for later.

Thanks to the classification model of DSM/StarBrowser, studying an arbitrary subset of a system is easy. We create a new DSMView and populate it with the four packages by dragging and dropping. Now we have limited the scope of our investigation, we can use the navigation features of DSMBrowser to quickly learn more about the system. By collapsing and expanding different modules, different cross-sections of the system can be visualised. For example, we could end up with the visualisation shown by figure 6.2 on the next page.

Having drilled down the level of classes and methods we can now start to investigate the system in more detail. Object-orientation demands that objects have intrinsic responsibilities but cooperate to implement the functionality of the system. With such principles in mind, we can expect that classes, or methods, with high incoming dependency values are relatively more important for the functioning of the system than others. Moreover, classes or methods which lack all incoming dependencies are likely to contain only "dead" code.

[2] Right-clicking on the `Namespace` package and selecting the "Inspect all dependencies to here" option from the context menu would reveal that the package is indeed only targeted by namespace dependencies.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
[+]DSM - Model	1	*	12															11	1			33
[+]DSM - StarBrowser2 Extensions	2	63	*																	6		7
[-]DSM - GUI	3			*																		
|--[-]DependencyCell	4				*											1						
| |--contentTypeString (DependencyCell)	5					*																
| |--dependencies (DependencyCell)	6						*															
| |--dependencies: (DependencyCell)	7							*														
| |--fromModuleCell (DependencyCell)	8								*													
| |--fromModuleCell: (DependencyCell)	9									*												
| |--initialize (DependencyCell)	10	1					1				*											
| |--menu (DependencyCell)	11	8										*				3					5	13
| |--printString (DependencyCell)	12	4											*								2	
| |--toModuleCell (DependencyCell)	13																					
| |--toModuleCell: (DependencyCell)	14																					
|--[+]DSMCell	15																				1	
|--[+]DSMCellList	16																					
|--[+]DSMTableInterface	17	18			1											2	8	*			1	
|--[+]DSMViewEditor	18	17										1				5	3	7	*		1	3
|--[+]DSMViewEditorShell	19																		2	*		
|--[+]ModuleCell	20	47										1				5	3	3	3		*	4
[+]DSM - Dependency Analysis	21	126																				*

Inspect dependencies
From module...
To module...
Run analysers...
Run advisors...
Inspect module
Inspect menu (DependencyCell)
Browse to menu (DependencyCell)
Inspect all dependencies from here
Inspect all dependencies to here

Figure 6.2: Exploring the implementation of DSMBrowser – Package subset

High outgoing dependency values are an indication of the entities that tie the system together, especially when dependencies cross the borders of surrounding entities. For instance, a class with many dependencies to classes in other packages is likely to be an important junction in the system.

Cleary, dependency values in the visualisation provide useful insights. While studying source code remains important to gain a full understanding of the system, the dependency values can act as a guide to explore the source code in a pragmatic way.

Guided by dependency values we can now browse through the system to find code entities that require further investigation. For instance, we may notice that the `menu` method has relatively more outgoing dependencies than the other methods of the `DependencyCell` class. For example it has 8 inter-package dependencies to the `Model` package. We might want to look at the source code of the `menu` method to find out what is going on. On the other hand, we can also inspect the dependencies themselves. To investigate selected code entities or sets of dependencies we can use the inspection features of DSMBrowser which are offered in context menus, as show in figure 6.2. These features allow us to move back and forth between the DSM visualisation and Refactoring Browser [61, 11] windows – to look at source code – or Trippy inspector [106] windows – to inspect dependencies or modules.

By continuing to explore the system in this fashion we can quickly learn to find our way around its principal components and gain understanding of its inner workings.

6.3 Dependency Management

Software developers, especially project leaders and software designers, deal with a wide variety of responsibilities related to dependency management. These responsibilities range from non-intervening coordination tasks to interventions in design or implementation. In this section we demonstrate how DSMBrowser can assist developers in such tasks.

In order to support coordination and communication among involved parties, dependency management requires that efforts are made to know and to keep track of implementation-level dependencies, throughout the lifecycle of a software product. In the case study in 6.3.1 we show how DSMBrowser can support these efforts. As an example we look at tracking dependencies to a third-party framework.

Dependency management also requires that opportunities for improved modularisation are detected, evaluated and pursued at various development stages. Pursuing such opportunities, through interventions in design or implementation, results in dependency minimisations or eliminations. In the case studies in 6.3.2, we show how the metaprogramming features of DSMBrowser can help developers to find and evaluate modularisation opportunities in the form of local architectural restructurings.

6.3.1 Tracking Framework Dependencies

Context
In section 5.2 of Chapter 5 we stated that dependency management is all the more important when parties with different (commercial) interests – such as other companies – are involved. As an illustration, we pointed out that depending on a third-party framework can be like tracking a moving target. In this scenario, we return to that example to demonstrate how DSMBrowser can help developers to keep track of framework dependencies.

Many companies use a framework supplied by a large industry player – for instance, Microsoft's .NET framework [98] or Sun's J2EE platform [102] – to build their own software products. These client companies, especially the smaller ones, are unlikely to have a big influence on future development of the framework. As a result, it may be unavoidable that new versions of the framework introduce changes that are incompatible with the applications that depend on it. Of course, the client companies can choose to stick with an older version of the framework, but that way they cannot benefit from improvements and bugfixes provided by new versions. Consequently, sooner or later at least part of the dependant applications will be migrated to a new version.

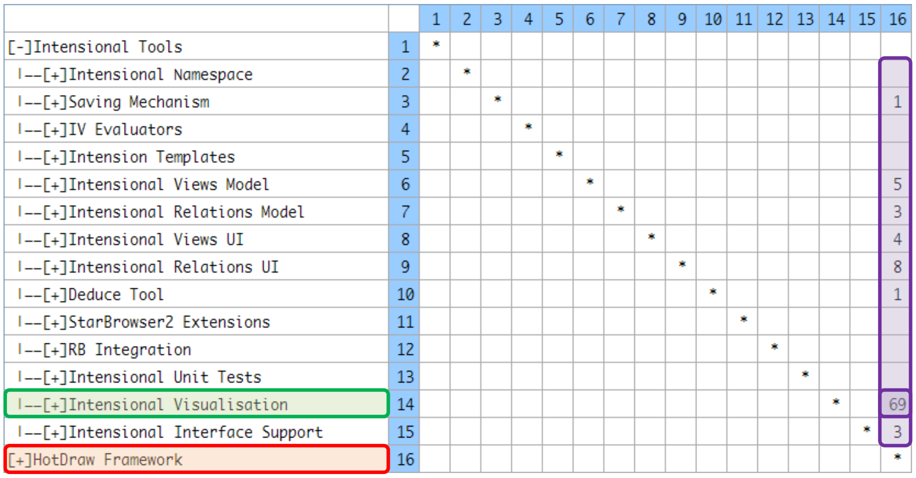
Usually the framework supplier provides migration support to its clients – at the very least, breaking changes in new framework versions will be documented – but in the end, it will be up to the programmers of the client companies to solve incompatibility problems on the implementation level. This requires insight into the locations and nature of dependencies to the framework.

Even when a client company does have a say, or is otherwise granted participation in the development of the framework, incompatibilities can only be avoided through intense communication and coordination, based on precise knowledge about implementation-level dependencies (e.g.: which types of dependencies occur how frequently in which locations and why).

We now illustrate how DSMBrowser can assist the users of a third-party frameworks to acquire such knowledge by explicitly tracking implementation-level dependencies to the framework. As an example, we investigated how the IntensiVE tool suite [30, 49, 48] depends on the HotDraw framework [10].

Usage Scenario

Figure 6.3 shows the DSM diagram we generated. We used two dependency filters in the process, one to filter out common message sends and another one to hide all internal dependencies among the packages of the `Intensional Tools` bundle. The former basically performs a kind of noise reduction on our measurements by throwing out unreliable dependencies[3]. The latter helps to focus on what is important here, namely the dependencies to the `HotDraw framework` package.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
[-]Intensional Tools	1	*															
|--[+]Intensional Namespace	2		*														
|--[+]Saving Mechanism	3			*													1
|--[+]IV Evaluators	4				*												
|--[+]Intension Templates	5					*											
|--[+]Intensional Views Model	6						*										5
|--[+]Intensional Relations Model	7							*									3
|--[+]Intensional Views UI	8								*								4
|--[+]Intensional Relations UI	9									*							8
|--[+]Deduce Tool	10										*						1
|--[+]StarBrowser2 Extensions	11											*					
|--[+]RB Integration	12												*				
|--[+]Intensional Unit Tests	13													*			
|--[+]Intensional Visualisation	14														*		69
|--[+]Intensional Interface Support	15															*	3
[+]HotDraw Framework	16																*

Figure 6.3: Tracking dependencies from IntensiVE (Intensional Tools) to the HotDraw framework

Clearly, the dependencies to classes of the `HotDraw framework` package are concentrated in the `Intensional Visualisation` package, which is not surprising since HotDraw is a

[3] Common message sends (e.g.: `new`, `initialize`, `release`, `add:`, etc.) cause a lot of false positives due to the dynamical typing system of Smalltalk and the naivety of the of the dependency target finding heuristic in the current version of DSMBrowser.

framework for drawing 2D graphics. However, the discovery of the few dependencies that originate from other packages – which do not deal with graphical visualisation – could be of greater importance, as the IntensiVE developers might not expect them there.

By expanding modules we can easily drill down to find out which classes, or even methods, are involved on both sides of the dependencies. On the one hand, tracking the sources of the dependencies tells us exactly which parts of IntensiVE depend on HotDraw functionality and how they do so. On the other hand, tracking the targets informs us about which specific parts of the framework are used by IntensiVE.

This kind information is of great value to both parties for numerous reasons, for instance:

- it is helpful when IntensiVE needs to be changed to work with a new version of HotDraw, as all the HotDraw-dependant locations are known;
- it makes documentation about changes in a new HotDraw version – supplied by the creators of the framework – a lot more useful, because the documentation can be compared with a list of all parts of HotDraw IntensiVE depends on, to quickly find out if and where there are breaking changes;
- it provides a suitable basis for communication, cooperation and coordination between both parties (e.g.: the IntensiVE developers can accurately inform the HotDraw creators about which parts of the framework they would like to remain unchanged);
- it helps to define which parts of HotDraw (or other frameworks) should be included when IntensiVE is finalised and packaged as a product to be supplied to customers. At the same time, it also indicates which dependencies need to be eliminated if the framework cannot be included in the distribution (e.g.: due to licensing constraints).

6.3.2 Finding and Evaluating Modularisation Opportunities

In what follows we demonstrate how DSMBrowser can assist developers to improve the modularity of existing software systems or during the implementation phase of systems that in development. By exploiting the metaprogramming facilities of DSMBrowser, we have developed metaprograms that automate the manual exploration, pattern detection, analysis and evaluation process to find feasible opportunities for improved modularisation. The task of implementing opportunities that have been found worthwhile remains up the users.

Modularisation Opportunities

These opportunities are implementation-level changes that could potentially improve modularity through minimisation or even elimination of dependencies. Specifically, we focus on local architectural restructurings by means of common object-oriented strategies. We study two general types of local restructurings.

First, we consider how the introduction of local indirections within modules can lower external dependencies. These restructurings reroute, canalize or centralise dependencies to external modules via a newly created local indirection entity. That way, multiple external dependencies are replaced by a single one, originating from the indirection. The local indirection acts as an interface through which the external module can be accessed. As a result, changes in the external module are less likely to require changes in all dependent entities but only require changes in the indirection entity. To find feasible indirection opportunities we have implemented two fully functional metaprograms, one which deals with message send dependencies and another one for inheritance dependencies. We will demonstrate both in small case studies.

Secondly, we consider how dependencies to external modules can be entirely eliminated by locally duplicating code that implements required behaviour. These restructurings can effectively separate or decouple modules. While we have not implemented them ourselves, we present an approach to create metaprograms to find opportunities for the elimination of message send and inheritance dependencies through code duplication, starting from the indirection metaprograms.

Metaprograms

The metaprograms we wrote analyse user-selected sets of dependencies and infer indicative suggestions or even specific advice, with regard modularity enhancing restructurings they consider to be feasible. We wrote these metaprograms in plain Smalltalk, using the analysis and metaprogramming facilities of DSMBrowser (the DSM Model API, dependency filters and the generic metaprogram classes) which we discuss in Chapter 5 (5.4.5).

Our metaprograms serve as a proof of concept, rather than universally applicable solutions. They use overly-general or naive assumptions and crisp threshold parameters to evaluate the feasibility of modularisation opportunities. While the current metaprograms can produce meaningful modularisation suggestions in real-world situations – as we will demonstrate – users will have to fine-tune the parameters and implement additional, domain specific decision rules, to apply them in specific, mission-critical environments.

Assumptions

In our metaprograms we make the assumption that the user wants to minimise dependencies which cross the boundaries of packages. In other words, the level of packages is considered to be the "separation point": the place where subsystems meet (e.g.: own code versus third-party code) and where strong dependencies are unwanted. It would however be trivial to change the metaprograms to lift the separation point to the bundle level or to drop it to the class level.

To illustrate the architectural restructurings we propose, we use UML [103] class diagrams with simplified "before" and "after" situations involving a package A and a package B. In these diagrams:

- The studied dependencies originate from package A and target package B;
- Package A considered to contain code which the user has full control over;
- Dependencies to package B are considered unwanted, for instance because it is controlled by a third party;
- In the "before" situation the packages contain 1 or 3 classes each. The number of classes per package should be interpreted as follows:
  - when a package X contains 1 class that should be read as "... *few* classes in package X ...",
  - when a package Y contains 3 classes that should be read as "... *many* classes in package Y ...",

  where the definitions of *few* and *many* are governed by user-configurable threshold parameters of the metaprograms.

6.3.2.1 Minimising Dependencies with Indirections

6.3.2.1.1 Message Send Dependencies

Restructurings
In this case study we present a metaprogram that suggests local architectural restructurings to minimise message send dependencies. All message send dependencies are caused by method calls which request the execution of behaviour or perform data access. We cannot simply eliminate these dependencies without affecting the functionality of the software. However, message sends that cross module boundaries can be reduced by introducing local indirections.

Given the assumptions discussed above, our aim is to lower the number of *inter-package* message sends by introducing indirections in the package of origin. Such indirections can be introduced on the level of classes and on the level of methods.

On the level of classes, we can use simple design patterns [22] to introduce local indirection classes to reroute message sends. We distinguish four situations with regard to message sends from a package A to a package B, based on the number of involved classes:

- **Many-to-Few:** The message sends originate from *many* classes in package A and target *few* classes in package B.
  In this case the dependency from package A to package B can be lowered by applying

the Proxy design pattern [22]. The idea is to introduce a local proxy-class in package A, for each of the *few* classes which are targeted in package B. Each proxy-class should contain wrapper methods for all methods of the class of package B it represents and which are called from package A. That way all calls to package B can be rerouted via local proxies. As a result there will only be a single inter-package message send dependency for every targeted method. This is illustrated by the UML diagram in figure 6.4a;

- **Many-to-Many:** The message sends originate from *many* classes in package A and target *many* classes in package B.
  In this case the dependency from package A to package B can be lowered by applying the Facade design pattern [22]. The idea is to introduce a single facade-class in package A, which contains wrapper methods for all methods of the *many* classes in package B, which are called from package A. That way all calls to package B can be rerouted via the local facade. As a result there will only be a single inter-package message send dependency for every targeted method. This is illustrated by the UML diagram in figure 6.4b;

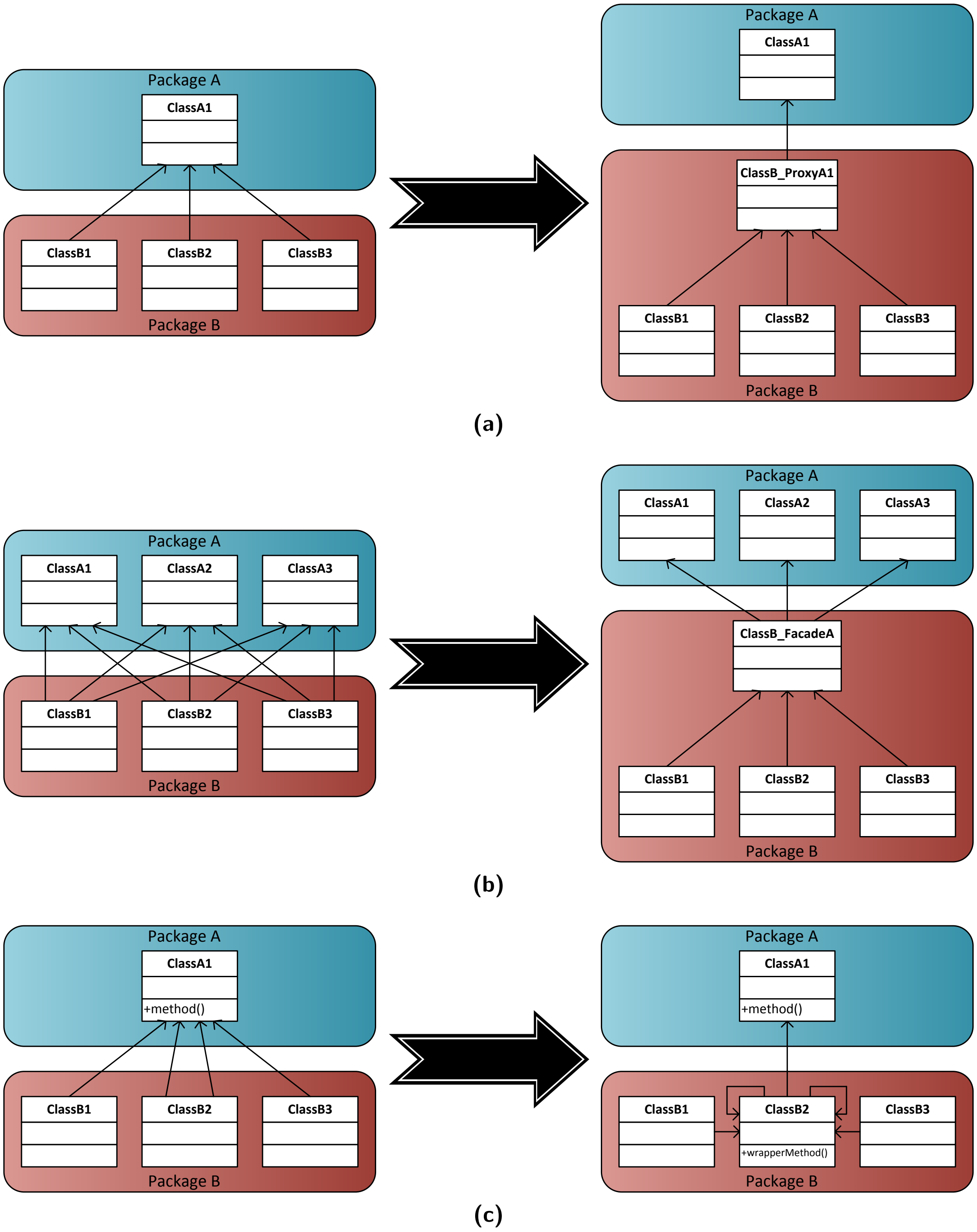
- **Few-to-Few:** The message sends originate from *few* classes in package A and target *few* classes in package B.
  In this case the introduction of indirections on the class-level is not useful;

- **Few-to-Many:** The message sends originate from *few* classes in package A and target *many* classes in package B.
  In this case the introduction of indirections on the class-level is not useful.

In the cases where there is no useful class-level indirection possible, method-level indirections can be considered to minimise message sends from a package A to a package B. Method-level indirections are simple, individual wrapper methods which are introduced into existing classes of package A, as illustrated by the UML diagram in figure 6.4c. This way the number of inter-package message send dependencies can be lowered whenever there are more messages being send than methods being targeted. The location where the wrapper method should go is best decided by considering the sources of the calls to the wrapped method. We propose to select the class which accounts for the most of those calls as the ideal place to insert the wrapper method.

Context

As an example we look at the message send dependencies from the IntensiVE tool suite [30, 49, 48] to SOUL [79, 76]. The implementation of IntensiVE depends heavily on SOUL and there is no way that local architectural restructurings can completely change that. However, the introduction of indirections can significantly lower the dependencies. Moreover, such indirections provide a local interfaces for SOUL components and thereby limit the effect of changes in SOUL for dependent components in IntensiVE.



(a)

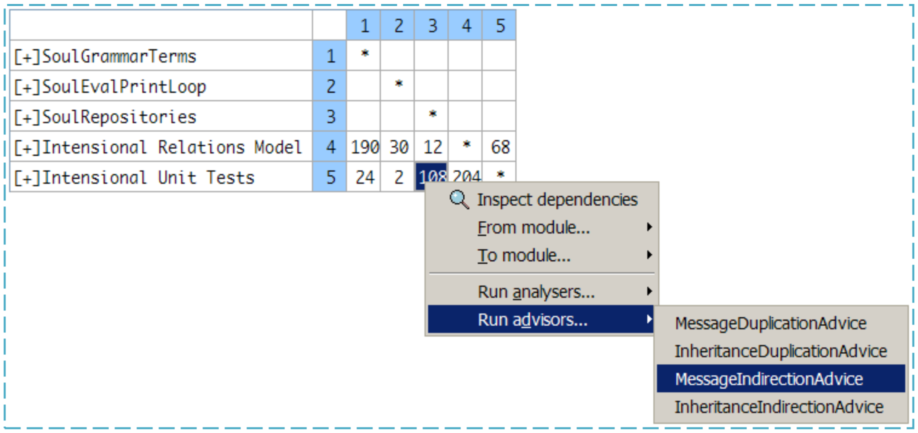
(b)

(c)

Figure 6.4: Lowering inter-package message send dependency by introducing indirections, by means of a proxy class (a), a facade class (b) and an individual wrapper method in an existing class (c)

Usage Scenario

First we created a new DSMView classification in StarBrowser and we populated it with all the packages of IntensiVE and those of SOUL. By filtering non-message send dependencies and dependencies caused by common messages we got an overview of the inter-package message send dependencies from IntensiVE to SOUL. Their number varied greatly for different pairs of packages. To limit the scope of our investigation we decided to create a second DSMView to look only at 2 packages of IntensiVE and 3 packages of SOUL which form the pairs with the strongest dependency. The result is shown by figure 6.5.



		1	2	3	4	5
[+]SoulGrammarTerms	1	*				
[+]SoulEvalPrintLoop	2		*			
[+]SoulRepositories	3			*		
[+]Intensional Relations Model	4	190	30	12	*	68
[+]Intensional Unit Tests	5	24	2	108	204	*

Figure 6.5: Running the MessageIndirectionAdvice metaprogram on a set of dependencies; Box 6.3 shows the resulting method-level advice for this set

To find out if and where indirections could be introduced to lower inter-package message send dependencies, we applied the MessageIndirectionAdvice metaprogram.

MessageIndirectionAdvice is a subclass of DependencyAdvisor, the generic superclass for metaprograms which produce suggestions, with regard to modularisation opportunities, for user-supplied sets of dependencies (see 5.4.5). These metaprograms produce results in the form of verbose textual reports.

Specifically, MessageIndirectionAdvice searches places where class-level and method-level indirections – as we discussed above – make sense to minimise inter-package message send dependencies. First, it uses filters to restrict the set of dependencies to message send dependencies, and only those that are not caused by common messages. Next, the dependencies are stored in a DependencyDictionary object. Then, it searches and evaluates indirection locations, using simple procedural logic and querying methods of the DependencyDictionary and other DSM Model classes (see 5.4.3 and 5.4.5). The process is governed by user-configurable parameters – for instance the *few* and *many* thresholds. The metaprogram can produce both class-level and method-level advice.

DSMBrowser dynamically adds entries for new metaprograms – subclassed from the generic metaprogram classes – to the context menu that is shown when a user right-clicks on a dependency value in the DSM visualisation, as shown by figure 6.5. That way it is easy to run metaprograms on specific dependency sets, directly from the GUI.

We applied MessageIndirectionAdvice to sets of dependencies from packages of IntensiVE to packages of Soul. We now show some of the reports we generated. In box 6.1 we see a class-level advice which suggests to introduce proxy classes to lower the dependency from `Intensional Relations Model` to `SoulEvalPrintLoop`.

```
Message sends from package Intensional Relations Model
              to package SoulEvalPrintLoop: 30

 Analysing for Class level advice...
  The message sends originate from 7 classes and target 4 classes.

  Advice: The degree of dependency from package Intensional Relations Model
          to package SoulEvalPrintLoop can be lowered by introducing
          4 PROXY classes in package Intensional Relations Model which
          represent classes EmptyEvaluator, Results, Evaluator, Binding
          of package SoulEvalPrintLoop.
```

Box 6.1: Message sends indirection advice (class-level) for dependencies from `Intensional Relations Model` to `SoulEvalPrintLoop`

Box 6.2 shows another class-level report, this one suggests to use a facade class to lower the dependency from `Intensional Relations Model` to `SoulGrammarTerms` package.

```
Message sends from package Intensional Relations Model
              to package SoulGrammarTerms: 190

 Analysing for Class level advice...
  The message sends originate from 8 classes and target 10 classes.

  Advice: The degree of dependency from package Intensional Relations Model
          to package SoulGrammarTerms can be lowered by introducing a
          FACADE class in package Intensional Relations Model which
          offers a local interface to the functionality provided by the 10
          classes from package SoulGrammarTerms to the 8 classes
          from package Intensional Relations Model which use it.
```

Box 6.2: Message sends indirection advice (class-level) for dependencies from `Intensional Relations Model` to `SoulGrammarTerms`

Finally, box 6.3 on the next page, shows a method-level advice report which suggests locations for wrapper methods to lower the dependency from `Intensional Unit Tests` to `SoulRepositories`.

```
Message sends from package Intensional Unit Tests
              to package SoulRepositories: 108

 Analysing for Method level advice...
  The message sends originate from 44 methods
              and target 3 methods in total.
  3 methods are targeted more than once.
  There are more message sends (108) than targeted methods (3)!

  Advice: The degree of dependency from package Intensional Unit Tests to
          package SoulRepositories can be lowered by introducing
          individual WRAPPER methods.
          The following list suggests a potentially suitable location
          (a class in package Intensional Unit Tests) for a wrapper method for
          each targeted method (with its class shown in parentheses):
           - indirection for assert:  (LogicRepository), which is targeted
             by 98 message sends, could be placed in class
             IntensionalRootTest.
           - indirection for name (LogicRepository), which is targeted by 6
             message sends, could be placed in class IntensionalRootTest.
           - indirection for removeLayer:  (LogicRepository), which is targeted
             by 4 message sends, could be placed in class IntensionalRootTest.
```

Box 6.3: Message sends indirection advice (method-level) for dependencies from Intensional Unit Tests to SoulRepositories

6.3.2.1.2 Inheritance Dependencies

Restructurings

In this case study we present a metaprogram that suggests local architectural restructurings to minimise inheritance dependencies. An inheritance dependency represents the relation from a subclass to its superclass. We cannot simply eliminate these dependencies without affecting the functionality of the software. However, inheritance relations that cross module boundaries can be reduced by introducing local indirections.

Given the assumptions discussed above, our aim is to lower the number of *inter-package* inheritance relations, by introducing indirections in the package of origin.

Whenever multiple classes in a package A inherit from single superclasses in a package B, the inter-package inheritance dependencies can be reduced by introducing a local intermediary superclasses in package A. An intermediary superclass inherits from the original superclass and acts as a local superclass for the subclasses of the original superclass in package A. This is illustrated by the UML diagram in figure 6.6 on the next page.

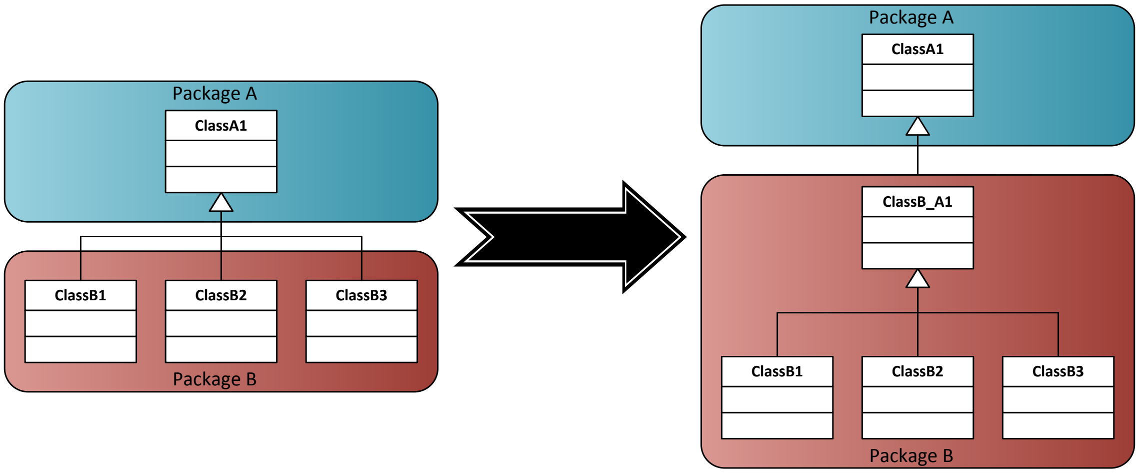


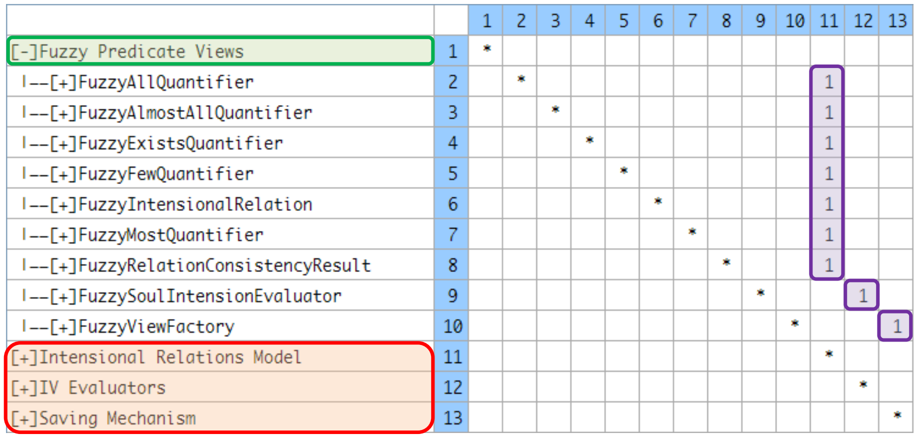
Figure 6.6: Lowering inter-package inheritance dependency by introducing an intermediary superclass

Context

As an example we look at the inheritance dependencies from the Fuzzy Intensional Views software [52] to the IntensiVE tool suite [30, 49, 48].

Usage Scenario

First, we created a new DSMView classification in StarBrowser and populated it with the packages of Fuzzy Intensional Views and IntensiVE. To get an overview of the inter-package inheritance dependencies we used a filter to remove all other dependencies from the DSM. The inheritance relations mainly originated from one package of Fuzzy Intensional Views and targeted classes in three packages of IntensiVE. Next we created a second DSMView to focus on just those packages, figure 6.7 shows the result.



		1	2	3	4	5	6	7	8	9	10	11	12	13
[-]Fuzzy Predicate Views	1	*												
I--[+]FuzzyAllQuantifier	2		*									1		
I--[+]FuzzyAlmostAllQuantifier	3			*								1		
I--[+]FuzzyExistsQuantifier	4				*							1		
I--[+]FuzzyFewQuantifier	5					*						1		
I--[+]FuzzyIntensionalRelation	6						*					1		
I--[+]FuzzyMostQuantifier	7							*				1		
I--[+]FuzzyRelationConsistencyResult	8								*			1		
I--[+]FuzzySoulIntensionEvaluator	9									*			1	
I--[+]FuzzyViewFactory	10										*			1
[+]Intensional Relations Model	11											*		
[+]IV Evaluators	12												*	
[+]Saving Mechanism	13													*

Figure 6.7: Inheritance dependencies from classes of the `Fuzzy Predicate Views` package to three packages of the IntensiVE tool suite

The DSM shows that there are 9 inheritance relations from the `Fuzzy Predicate Views` package to the three packages of the IntensiVE tool suite.

To find out if and where indirections could be introduced to lower inter-package inheritance dependencies, we applied the InheritanceIndirectionAdvice metaprogram.

InheritanceIndirectionAdvice is also a subclass of DependencyAdvisor dependencies (see 5.4.5), and is thus largely analogous to MessageIndirectionAdvice. It searches places where it makes sense to introduce an intermediary superclass – as we discussed above – to minimise inter-package inheritance dependencies.

We applied this metaprogram to the dependencies from the `Fuzzy Predicate Views` package to each of the three packages of the IntensiVE tool suite. Box 6.4 shows the results.

```
Inheritance relations from package Fuzzy Predicate Views
                      to package Intensional Relations Model: 7
 Advice:
  The degree of dependency from package Fuzzy Predicate Views to
  package Intensional Relations Model can be lowered by introducing
  a new subclass of class AbstractQuantifier of package
  Intensional Relations Model in package Fuzzy Predicate Views to
  act as a local, INTERMEDIARY SUPERCLASS for classes FuzzyAllQuantifier,
  FuzzyAlmostAllQuantifier, FuzzyMostQuantifier, FuzzyFewQuantifier,
  FuzzyExistsQuantifier which are currently directly subclassed from
  class AbstractQuantifier of package Intensional Relations Model.

Inheritance relations from package Fuzzy Predicate Views
                      to package IV Evaluators: 1
 No advice for these packages.

Inheritance relations from package Fuzzy Predicate Views
                      to package Saving Mechanism: 1
 No advice for these packages.
```

Box 6.4: Inheritance Indirection Advice Report

The report shows that the 7 inheritance dependencies from the `Fuzzy Predicate Views` package to the `Intensional Relations Model` package, all target the same superclass (`AbstractQuantifier`). Hence, an intermediary superclass could bring the inheritance dependencies between those packages down from 7 to 1. The `Fuzzy Predicate Views` package only has single inheritance dependencies to the other two packages of IntensiVE, so obviously those dependencies cannot be lowered any further by means of indirections.

6.3.2.2 Eliminating Dependencies with Code Duplication

Duplication of code is generally considered as bad practice because it the hampers maintenance and evolution of software systems. Indeed, each new software development paradigm that has been introduced has provided new abstraction mechanisms that allow us to achieve increased modularity and reduced code duplication in software implementations.

However, there are situations where code duplication is a drastic, yet viable, option to deal with dependency related issues. For instance, when a software module depends on a small part of a large, external module which is otherwise obsolete, outdated or too big to include in distributions, it might be better to locally duplicate the code of that small part. That way the dependency to the external module can be entirely eliminated.

The current version of DSMBrowser does not include metaprograms to find and evaluate places where dependencies can be eliminated through code duplication. However, the provided APIs should suffice to create such programs. Furthermore, the programs that deal with indirections, which we discussed above, could serve as a starting point. The major difference would be that checking for possible code duplication opportunities requires the inspection of second, and higher, degree dependencies, further away from the source of the dependency that is being considered for elimination. This is necessary to determine whether or not the targeted entity is sufficiently independent from its surroundings to be copied in isolation or with a limited number of supporting entities.

Code duplication can be used to eliminate both message send and inheritance dependencies.

6.3.2.2.1 Message Send Dependencies

Message send dependencies from package A to package B can be eliminated by locally duplicating the behaviour of the called methods of classes of package B. Duplicated methods can be inserted into existing classes of package A, or new classes can be created to accommodate them.

Not all methods can be duplicated in isolation of rest of the class they are part of. Therefore, it may be necessary to duplicate the whole class, possibly with unneeded methods and data structures stripped out. But the class itself could also be closely dependent of other classes of package B, which would need to be duplicated as well, and so on. Clearly, it is not trivial to decide which duplication efforts are worthwhile.

The duplication of a class to eliminate message send dependencies is illustrated by the UML diagram in figure 6.8, on the next page.

6.3.2.2.2 Inheritance Dependencies

Inheritance dependencies from package A to package B can be eliminated by duplicating the targeted superclasses in package B as local classes in package A. Again, it is not trivial to decide which classes can be duplicated without requiring too many additional classes to be copied as well. The duplication of a superclass to eliminate inheritance dependencies is illustrated by the UML diagram in figure 6.9, on the next page.

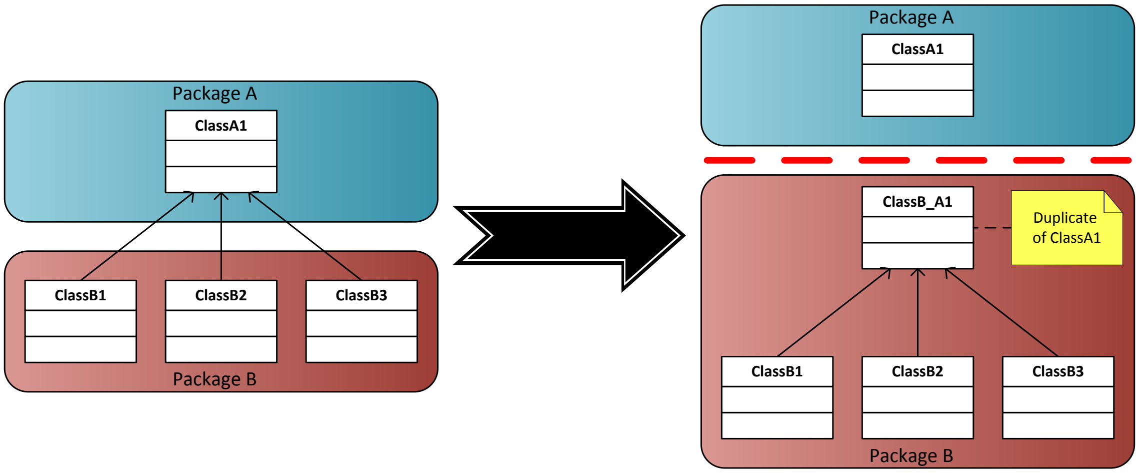


Figure 6.8: Lowering inter-package message send dependency by introducing code duplication

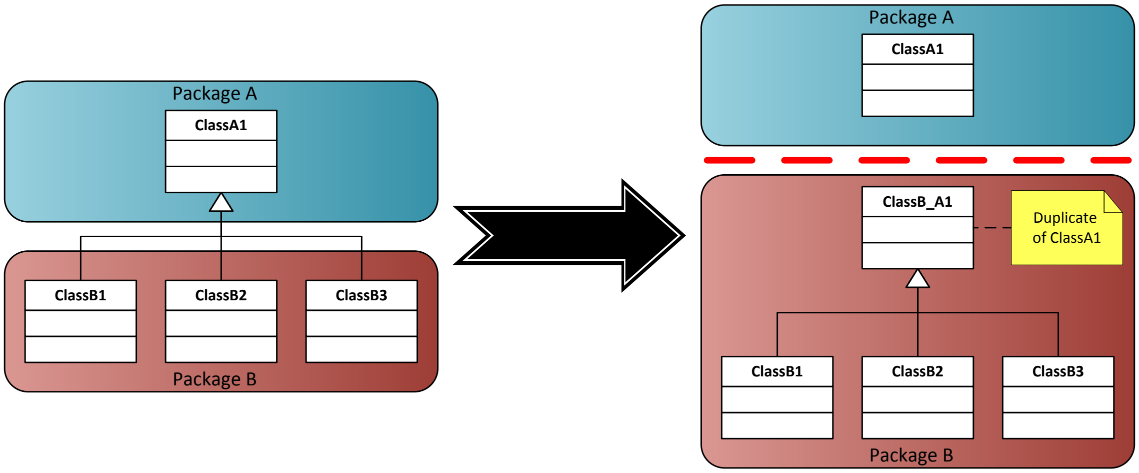


Figure 6.9: Lowering inter-package inheritance dependency by introducing code duplication

6.4 Conclusions

The case studies we presented in this chapter demonstrate, by means of real-world examples, how the diverse features of DSMBrowser, or future similar tools, can effectively assist software developers in common tasks.

Our exploration case study showed how the tool can help developers to explore and familiarise themselves with large amounts of unknown source code. The tree-based DSM visualisation proved to be an excellent guide to explore systems in a pragmatic way.

We also demonstrated at large how DSMBrowser facilitates tasks with regard to dependency management. First, we explained how the tool can be used to generate helpful overviews of dependencies to external systems, such as third-party frameworks. These overviews can serve as a basis for communication and coordination among involved parties. Next, we showed how the metaprogramming facilities of DSMBrowser can be used to automate the process of finding and evaluating different kinds of modularisation opportunities, in the form of local architectural restructurings.

These results validate the claims we made in the previous chapter. Most notably they emphasise that the concept of an IDE-integrated source-code browser which provides tree-based DSM visualisations and metaprogramming-based analysis facilities, is indeed a valuable contribution to the OOSD field.

In Chapter 7 (section 7.3) we formulate ideas for future enhancements to DSMBrowser and for more sophisticated metaprograms.

Chapter 7

Conclusions

7.1 Summary

In this dissertation we investigated how Design Structure Matrices (DSMs) and related techniques can be applied in the context of software development. The premise of our work is that DSM diagrams, through their focus on the modular structure of systems and the explicit depiction of – the distribution and nature of – dependencies, offer interesting opportunities for innovations on two distinct fronts.

First we have provided an elaborate introduction to DSMs, their origins and application and related techniques such as NOV. Next, we presented the two angels of research we have pursued.

The first part of our research covered an exploration of the combination of DSMs and the Net Option Value (NOV) model as a methodology for quantitative assessment of modularity in software. First we introduced *DSM+NOV Tool*, a novel software tool intended to facilitate experimentation with this methodology. Next, we presented an evaluation of the NOV model as a modularity metric for software design, based on experiments on aspect-oriented and object-oriented design pattern implementations. We concluded that the current level of understanding of the NOV model regrettably does not warrant its application it this context.

The second part of our work constituted an investigation of the merits of DSM diagrams as a basis for a novel kind of support tools for software development. We analysed the possibilities and requirements for a DSM-based support tool for Object-Oriented Software Development (OOSD) and summarised our findings in an extensive rationale, illustrated by a prototype implementation called *DSMBrowser*. Next, we presented a number of real-world case studies which demonstrate how our prototype, or future similar tools, can assist software developers in common tasks related to source code exploration and dependency management.

7.2 Contributions

The work we presented in this dissertation offers the following contributions:

- In Chapter 2 we provided a comprehensive introduction to DSMs and related techniques such as NOV. This text offers readers an essential crash course which prepares them for passive and active use of DSM diagrams for both research and professional purposes;

- In Chapter 3 we introduced a novel software tool which facilitates the application of the DSM+NOV methodology for design assessments, as well as exploration of the methodology itself. The tool introduces an innovative technique to assess systems using NOV and a combination of two DSMs which focus on different hierarchical levels of the studied system;

- In Chapter 4 we investigated the DSM+NOV methodology as a technique for qualitative assessment of software. We introduced a novel approach for NOV measurement and a new assumption for the complexity parameter of the NOV model, both specifically aimed at analysis of software implementations. We concluded this evaluation with a series of observations that led us to strongly question the applicability of the NOV model as a modularity metric for software design;

- In Chapter 5 presented a rationale for a novel DSM-based support tool for OOSD and a prototype implementation of such a tool. As part of the rationale we formulated essential requirements for a dependency management tool for OOSD and we proposed an approach which combines tree-based DSM visualisations and metaprogramming-based analysis facilities in an extendable, IDE-integrated source-code browser. In our prototype we successfully applied this approach to support software development in the Smalltalk language.

- Chapter 6 we discussed case studies which show how DSM-based source browsers can assist software developers in the exploration of unknown source code and in dependency management tasks. As a demonstration of the metaprogramming facilities of our prototype we presented metaprograms that automate the task of finding and evaluating modularisation opportunities based on local architectural restructurings.

7.3 Future Work

In this section we give an overview of future research directions we consider interesting for each of the two angels we followed in our work.

7.3.1 Evaluating Modularity with DSMs

Further research into the NOV model may enhance our understanding of its parameters and outcome and could then ultimately enable us to successfully use it in combination with numerical DSMs (NDSMs) and as software metric. However, due to difficulties we experienced in our experiments with the model, we have significant doubts concerning the chance of success for this research direction.

Therefore, we think it would be better to focus on developing a new quantitative model to assess modularity in design based on DSMs. Such a new model could retain some of the principles of NOV but it should be specifically designed to serve as a modularity metric for software design.

7.3.2 DSM-based Support Tools

We have a multitude of ideas with regard to new features for our *DSMBrowser* tool and/or future research into DSM-based support tools in general.

We think that it could be interesting to introduce a time aspect in DSM-based support tools. This would enable us to use DSMs to visualise the evolution of a software system, rather than working with static representations frozen in time. We could investigate these possibilities by extending our tool with a versioning concept. In connection with this idea it might be interesting to look at the Moose reengineering environment for Smalltalk [54].

Currently our metaprograms are limited to finding and evaluating modularisation opportunities based on relatively small local architectural restructurings. An obvious improvement we could focus on would be to devise more sophisticated metaprograms capable of finding and evaluating modularisation opportunities on a larger scale. For instance, we could try to detect situations where the introduction of design patterns [22] that encompass much larger parts of the system could effectively improve the modularity of the design as a whole. In fact, we could extend our source-code browser to become a full-blown refactoring tool which detects opportunities for refactorings based on "bad smells" [21, 31] and integrates with the refactoring framework of the Refactoring Browser [61, 11].

Until now our DSMBrowser metaprograms only find and evaluate modularisation opportunities, leaving the task of pursuing opportunities that have been found worthwhile to the developers themselves. In the future we could try to automate this third phase as well, by further exploiting the metaprogramming features of Smalltalk to let our metaprograms make actual changes to the implementation of the studied system. This would allow users to instantly see the effects of these changes in the DSM visualisation, without first having to resort to manual source code editing. Furthermore, if this idea is realised in combination with the time aspect idea, users would be able to compare "before" and "after" situations.

If we integrate DSMBrowser with the Refactoring Browser framework we could let our metaprograms trigger the automatic refactorings offered by that framework.

For now our metaprograms are written in a procedural and imperative style and in plain Smalltalk. In the future we could consider to extend DSMBrowser with support for declarative metaprogramming using a domain specific language. For instance, it might be interesting to investigate whether DSMBrowser can be linked to the SOUL framework [79, 76]. If we combine a declarative approach to metaprogramming with support for refactorings we could use the work of Muñoz Bravo [53] as a source of inspiration.

Last but not least, we would like to investigate if DSMBrowser can be used as a basis for a DSM-based aspect mining tool [28].

7.4 Conclusion

In this dissertation we to took the first steps towards the application of Design Structure Matrices (DSMs) in diverse aspects of software development.

On the one hand, our initial expectations with regard to the Net Option Value (NOV) model proved to be too optimistic. This is shown by the evaluation we presented, which clearly highlights the obstacles that hamper the application of the NOV model in the context of software development. Nevertheless, the tool be developed to support our experiments can be used for further research into NOV or for applications of DMS+NOV methodology outside software development.

On the other hand, the use of DSMs diagrams in support tools proved to be a real success, as demonstrated by the case studies we presented. Therefore, we are convinced that our rationale for DSM-based source code browsers and our prototype implementation are major contributions to the field.

Appendix A

The Q(k) distribution

The calculation of Net Option Value (NOV) simulates parallel experimentation on each constituent module of a design (see 2.3.6.2 in Chapter 2). The role of the $Q(k)$ distribution is to quantify the value of the best of k outcomes (resulting from k experiments) for each module.

$Q(k)$ is defined as the expected value of the highest realisation (best/maximum draw) of k independent draws from a standard normal distribution, as long as the realisation is greater than zero (so only for all positive values in the distribution)[1].

The formal definition of the distribution is given by equation A.4, where $N(x)$, given by equation A.5, is the cumulative distribution function and $n(x)$, given by equation A.6, the probability density function of the standard normal distribution. These functions are respectively the evaluation of equation A.1 and equation A.2 for the *standard* normal distribution (mean $\mu = 0$; standard deviation $\sigma = 1$). The Gauss error function, $erf(x)$, is defined by equation A.3.

Normal distribution

Cumulative distribution function:

$$F(x;\mu,\sigma) = \frac{1}{2}\left[1 + erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right] \tag{A.1}$$

Probability density function:

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\ e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{A.2}$$

Gauss error function

$$erf(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\ dt \tag{A.3}$$

$$Q(k) = k\int_0^\infty x\ [N(x)]^{k-1}\ n(x)\ dx \tag{A.4}$$

$$N(x) = \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right] \tag{A.5}$$

$$n(x) = \frac{1}{\sqrt{2\pi}}\ e^{-\frac{x^2}{2}} \tag{A.6}$$

[1] The distribution of the best of k realisations is common in statistics: it is the distribution of the "maximum order statistic of a sample of size k". However, in this case the expectation differs from the standard one, as it is taken only over the range of values above zero (see [5, 7], which cite [42] as a reference on order statistics in general).

Table A.1 tabulates the values of $Q(k)$ for $0 \leq k \leq 100$. On the next page, figure A.1 shows these values as a graph.

k	$Q(k)$
0	0.000000
1	0.398942
2	0.681037
3	0.888147
4	1.045756
5	1.169705
6	1.270073
7	1.353426
8	1.424153
9	1.485261
10	1.538865
11	1.586488
12	1.629251
13	1.668001
14	1.703387
15	1.735916
16	1.765993
17	1.793943
18	1.820032
19	1.844482
20	1.867475
21	1.889168
22	1.909692
23	1.929162
24	1.947674
25	1.965315
26	1.982158
27	1.998269
28	2.013707
29	2.028522
30	2.042761
31	2.056464
32	2.069669
33	2.082408
34	2.094713
35	2.106609
36	2.118123
37	2.129277
38	2.140091
39	2.150586
40	2.160777
41	2.170682
42	2.180316
43	2.189691
44	2.198822
45	2.207720
46	2.216395
47	2.224859
48	2.233121
49	2.241190
50	2.249074
51	2.256781
52	2.264319
53	2.271694
54	2.278914
55	2.285983
56	2.292909
57	2.299696
58	2.306351
59	2.312876
60	2.319278
61	2.325561
62	2.331728
63	2.337785
64	2.343733
65	2.349578
66	2.355323
67	2.360970
68	2.366524
69	2.371986
70	2.377359
71	2.382647
72	2.387852
73	2.392976
74	2.398022
75	2.402992
76	2.407889
77	2.412713
78	2.417467
79	2.422154
80	2.426774
81	2.431331
82	2.435824
83	2.440257
84	2.444630
85	2.448945
86	2.453204
87	2.457407
88	2.461557
89	2.465654
90	2.469700
91	2.473697
92	2.477644
93	2.481544
94	2.485397
95	2.489204
96	2.492967
97	2.496687
98	2.500364
99	2.503999
100	2.507594

Table A.1: Values of $Q(k)$ for $0 \leq k \leq 100$, rounded to the nearest 10^{-6}

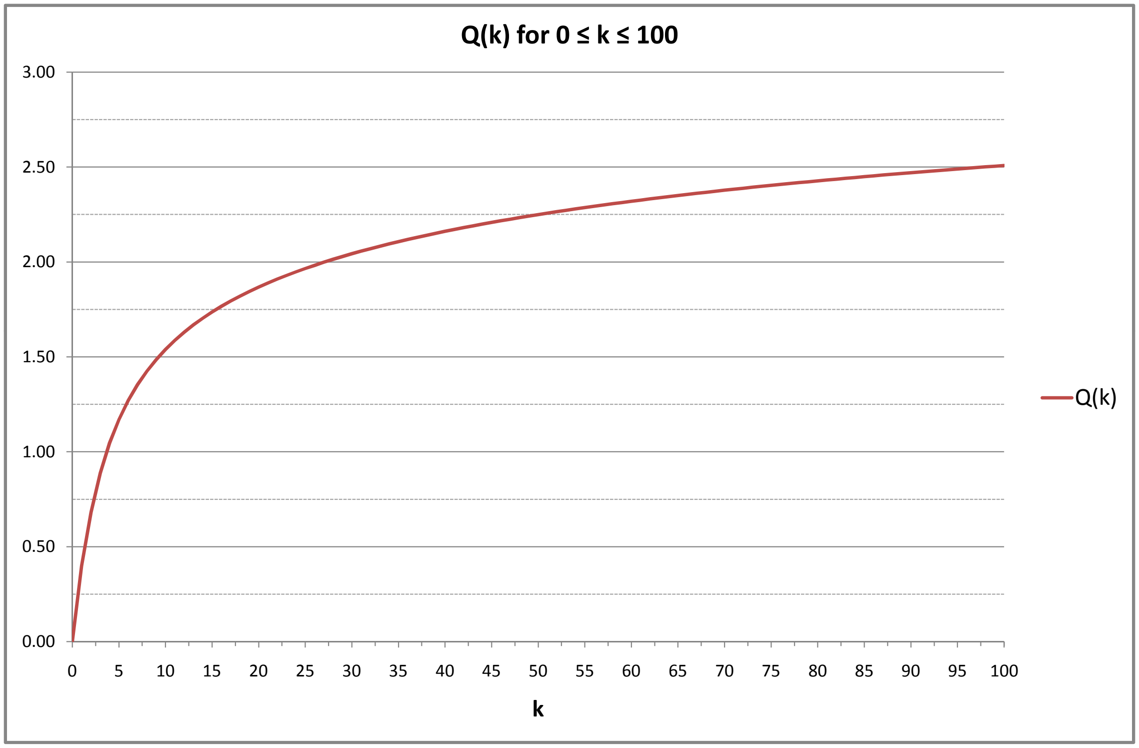


Figure A.1: Graphed values of $Q(k)$ for $0 \leq k \leq 100$

Bibliography

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, et al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, USA, 1977.
ISBN: 0-195-01919-9.

[2] Sushil Krishna Bajracharya. DeMatrix, 2006.
URL: http://mine7.ics.uci.edu/repo2/dsm.html.

[3] Sushil Krishna Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Videira Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, USA*, pages 681–682, New York, NY, USA, October 2006. ACM Press.
ISBN: 1-59593-491-X.
DOI: 10.1145/1176617.1176671.

[4] Sushil Krishna Bajracharya, Trung Chi Ngo, and Cristina Videira Lopes. On using Net Options Value as a Value Based Design Framework. In Kevin J. Sullivan, editor, *EDSER '05: Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research*, pages 1–3. ACM Press, New York, NY, USA, May 2005. EDSER '05 took place on May 15, 2005 as a workshop part of ICSE '05, the 27th International Conference on Software Engineering, May 15-21, 2005, St. Louis, MO, USA.
ISBN: 1-59593-118-X.
DOI: 10.1145/1083091.1083104.

[5] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Volume 1: The Power of Modularity*. MIT Press, Cambridge, MA, USA, March 2000.
ISBN: 0-262-02466-7.
URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3606.

[6] Carliss Y. Baldwin and Kim B. Clark. The Option Value of Modularity in Design: An Example from "Design Rules, Volume 1: The Power of Modularity". Harvard NOM Working Paper No. 02-13; Harvard Business School Working Paper No. 02-078, Harvard Business School, Boston, MA, USA, May 2002.

URL: http://ssrn.com/abstract=312404,
DOI: 10.2139/ssrn.312404.

[7] Carliss Y. Baldwin and Kim B. Clark. Modularity in the Design of Complex Engineering Systems. In Dan Braha Ali Minai and Yaneer Bar Yam, editors, *Complex Engineered Systems: Science Meets Technology*, Understanding Complex Systems, chapter 9, pages 175–205. New England Complex Systems Institute (NECSI) / Springer-Verlag, 2006. Article itself dates from January 2004.
ISBN: 3-540-32831-9, 978-3-540-32831-5.
URL: http://www.people.hbs.edu/cbaldwin/DR2/BaldwinClarkCES.pdf.

[8] Kent Beck and Ward Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical Report Technical Report No. CR-87-43, Tektronix, Inc., 1987. Presented at the OOPSLA'87 Workshop on Specification and Design for Object-Oriented Programming.
URL: http://c2.com/doc/oopsla87.html.

[9] Thomas A. Black, Charles H. Fine, and Emanuel M. Sachs. A Method for Systems Design Using Precedence Relationships: An Application to Automotive Brake Systems. Working Paper WP #3208-90-MS, Leaders for Manufacturing Program, MIT Sloan School of Management, Cambridge, MA, USA, October 1990.
URL: http://hdl.handle.net/1721.1/2324.

[10] John Brant et al. HotDraw, a two-dimensional graphics framework for VisualWorks Smalltalk, 1992.
URL: http://st-www.cs.uiuc.edu/users/brant/HotDraw.

[11] John Brant and Don Roberts. The Refactoring Browser. In Serge Demeyer and Jan Bosch, editors, *Object Oriented Technology – ECOOP'98 Workshop Reader: ECOOP'98 Workshops, Demos, and Posters, Brussels, Belgium, July 1998. Proceedings*, volume 1543/1998 of *Lecture Notes in Computer Science (LNCS)*, page 549. Springer-Verlag, Berlin/Heidelberg, Germany, July 1998.
ISBN: 3-540-65460-7, ISSN: 0302-9743.
URL: http://www.springerlink.com/content/krcpn19xpc3dw5gm.

[12] Johan Brichau and Theo D'Hondt. Introduction to Aspect-Oriented Software Development (white paper). Technical Report AOSD-Europe Deliverable D17, AOSD-Europe-VUB-02, Vrije Universiteit Brussel, August 2005.
URL: http://www.aosd-europe.net/deliverables/d17.pdf.

[13] Tyson R. Browning. Use of Dependency Structure Matrices for Product Development Cycle Time Reduction. In *Proceedings of the Fifth ISPE International Conference on Concurrent Engineering: Research and Applications, Tokyo, Japan, July 15-17, 1998*, pages 15–17, July 1998.

[14] Tyson R. Browning. "The Design Structure Matrix". In Richard C. Dorf, editor, *The Technology Management Handbook*, pages 103–111. Chapman & Hall/CRCnetBASE, Boca Raton, FL, USA, 1999.

[15] Tyson R. Browning. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, August 2001.

[16] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, USA, December 2004.
ISBN: 0-32124-587-3.
URL: http://www.informit.com/store/product.aspx?isbn=0321245873.

[17] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of Object-Oriented Software Design and Implementation. In *Proceedings of the International Symposium on Software Architectures and Component Technology (SACT 2000), Twente, The Netherlands*, January 2000.
URL: http://prog.vub.ac.be/Publications/2000/vub-prog-tr-00-03.pdf.

[18] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, Upper Saddle River, NJ, USA, October 1976.
ISBN: 0-13215-871-X, 978-0-132-15871-8.

[19] Steven D. Eppinger. Innovation at the Speed of Information. *Harvard Business Review*, 79(1):149–158, January 2001.
ISSN: 0017-8012.
URL: http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=3933461&site=ehost-live.

[20] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, Boston, MA, USA, October 2004.
ISBN: 0-321-21976-7.
URL: http://www.informit.com/store/product.aspx?isbn=0321219767.

[21] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, Boston, MA, USA, 1999.
ISBN: 0-201-48567-2.
URL: http://www.informit.com/store/product.aspx?isbn=0201485672.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Boston, MA, USA, October 1994.
ISBN: 0-201-63361-2.
URL: http://www.informit.com/store/product.aspx?isbn=0201633612.

[23] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Prentice Hall PTR, Indianapolis, IN, USA, third edition, June 2005.
ISBN: 0-321-24678-0, 978-0-321-24678-3.
URL: http://java.sun.com/docs/books/jls,
URL: http://www.phptr.com/bookstore/product.asp?isbn=0321246780.

[24] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, New York, NY, USA, 2002.
ISBN: 1-58113-471-1.
DOI: 10.1145/582419.582436.

[25] Jan Hannemann and Gregor Kiczales. Aspect-Oriented Design Pattern Implementations, 2004. Latest available version: v1.11.
URL: http://www.cs.ubc.ca/~jan/AODPs.

[26] Karim Jamal and Clinton Jenkins. Tool Evaluation – Lattix LDM. Mini Project by Team OverHEAD for course Analysis of Software Artifacts (School of Computer Science, Carnegie Mellon University), April 2006.
URL: http://www.cs.cmu.edu/~aldrich/courses/654/tools/overhead-lattix-06.pdf.

[27] Andy Kellens, Kris Gybels, Johan Brichau, and Kim Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *SPLAT! 2006 – Software Engineering Properties of Languages and Aspect Technologies (A workshop affiliated with AOSD '06, March 21, 2006; Bonn, Germany)*, 2006.
URL: http://aosd.net/workshops/splat/2006/papers/kellens.pdf.

[28] Andy Kellens and Kim Mens. A Survey of Aspect Mining Tools and Techniques. Project IWT 040116 "AspectLab" – Workpackage 6 – Deliverable 6.2.a, Programming Technology Lab, Vrije Universiteit Brussel and Département d'Ingénierie Informatique, Université Catholique de Louvain, June 2005.
URL: http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-16.pdf.

[29] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067/2006 of *Lecture Notes in Computer*

Science (LNCS), pages 501–525. Springer-Verlag, Berlin/Heidelberg, Germany, 2006.
DOI: 10.1007/11785477_28.

[30] Andy Kellens, Kim Mens, et al. IntensiVE, the Intensional View Environment, 2005.
URL: http://www.intensional.be.

[31] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Signature Series. Addison-Wesley Professional, Boston, MA, USA, August 2004.
ISBN: 0-321-21335-1, 978-0-321-21335-8.
URL: http://www.informit.com/store/product.aspx?isbn=0321213351.

[32] Gregor Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, December 1996.
ISSN: 0360-0300.
DOI: 10.1145/242224.242420.

[33] Gregor Kiczales. AspectJ^{TM}: aspect-oriented programming using Java^{TM} technology. Presentation held at *JavaOne Conference*, June 2000.
URL: http://www.parc.com/research/projects/aspectj/downloads/JavaOne-2000.ppt.

[34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072/2001 of *Lecture Notes in Computer Science (LNCS)*, pages 327–353. Springer-Verlag, Berlin/Heidelberg, Germany, 2001.
ISBN: 3-540-42206-4, ISSN: 0302-9743.
URL: http://www.springerlink.com/content/mc9xermkrav48ff1/?p=79ed4cdcf4834bd6a56e7c7969386875&pi=17.

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001. Special Issue on Aspect-Oriented Programming.
ISSN: 0001-0782.
DOI: 10.1145/383845.383858.

[36] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-Oriented Programming. In *DSL '97 – Proceedings of the first ACM SIGPLAN Workshop on Domain-Specific Languages*, University of Illinois Computer Science Report, pages 75–88, January 1997.
URL: http://www-sal.cs.uiuc.edu/~kamin/dsl.

[37] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet

Akşit and Satoshi Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming: 11th European Conference, Jyväskylä, Finland, June 9–13, 1997, Proceedings*, volume 1241/1997 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242. Springer-Verlag, Berlin/Heidelberg, Germany, June 1997.
ISBN: 3-540-63089-9, ISSN: 0302-9743.
DOI: 10.1007/BFb0053381.

[38] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, IN, USA, 2002.
ISBN: 0-672-32410-5, 978-0-672-32410-9.
URL: http://www.samspublishing.com/title/0672324105.

[39] Christian Koppen and Maximilian Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *EIWAS'04 – First European Interactive Workshop on Aspects in Software (September, 23-24, 2004. Berlin, Germany)*, August 2004.

[40] Robert E. Kraut and Lynn A. Streeter. Coordination in Software Development. *Communications of the ACM*, 38(3):69–81, 1995.
ISSN: 0001-0782.
DOI: 10.1145/203330.203345.

[41] Lattix, Inc. Example: A Dependency Model for Apache Ant, 2006.
URL: http://www.lattix.com/technology/antexample.php.

[42] Bernard W. Lindgren. *Statistical Theory*. Texts in Statistical Science. Chapman & Hall/CRC, 4th edition, October 1993.
ISBN: 0-41204-181-2, 978-0-412-04181-5.

[43] Cristina Videira Lopes. AOP: A Historical Perspective (What's in a Name?). In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 5, pages 97–122. Addison-Wesley, Boston, MA, USA, October 2004.
ISBN: 0-321-21976-7.
URL: http://www.informit.com/store/product.aspx?isbn=0321219767.

[44] Cristina Videira Lopes and Sushil Krishna Bajracharya. An Analysis Of Modularity In Aspect Oriented Design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development (March 14-18, 2005; Chicago, Illinois, USA)*, pages 15–26. ACM Press, New York, NY, USA, March 2005.
ISBN: 1-59593-042-6.
DOI: 10.1145/1052898.1052900.

[45] Cristina Videira Lopes and Sushil Krishna Bajracharya. Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. In Awais Rashid

and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880/2006 of *Lecture Notes in Computer Science (LNCS)*, pages 1–35. Springer-Verlag, Berlin/Heidelberg, Germany, February 2006.
ISBN: 978-3-540-32972-5, 3-540-32972-2, ISSN: 0302-9743.
DOI: 10.1007/11687061_1.

[46] Cristina Videira Lopes and Sushil Krishna Bajracharya. NOV Worksheet Template (noa-nov-template.xls), 2006.
URL: http://mondego.calit2.uci.edu/xwiki/bin/view/XWiki/DSM.

[47] Kent R. McCord and Steven D. Eppinger. Managing the Integration Problem in Concurrent Engineering. Working Paper WP #3594-93-MSA, Leaders for Manufacturing Program, MIT Sloan School of Management, Cambridge, MA, USA, August 1993.
URL: http://hdl.handle.net/1721.1/2484.

[48] Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR), 2006*, March 2006.
ISBN: 0-7695-2536-9, ISSN: 1052-8725.
DOI: 10.1109/CSMR.2006.29.

[49] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. The Intensional View Environment. In *Proceedings of the 21st IEEE International Conference on Software Maintenance – Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, pages 81–84, September 2005.
ISBN: 9-6346-0980-5.
URL: http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-31.pdf.

[50] Russell Miles. *AspectJ Cookbook*. O'Reilly Media Inc., Cambridge, MA, USA, 1st edition, December 2004.
ISBN: 0-596-00654-3, 978-0-596-00654-9.
URL: http://www.oreilly.com/catalog/aspectjckbk.

[51] Martin E. Modell. *A Professional's Guide to Systems Analysis*, chapter 6 - Project Planning. Mcgraw-Hill Software Engineering Series. McGraw-Hill Companies, New York, NY, USA, 2nd edition, June 1996.
ISBN: 0-07-042948-0, 978-0-07-042948-2.
URL: http://www.martymodell.com/pgsa2/pgsa06.html.

[52] Karlien Mollemans. Introducing Flexible Causal Links by Fuzzifying Intensional Views. Master's thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, Departement Informatica, Programming Technology Lab, Brussels, Belgium, September 2007.

[53] Francisca Muñoz Bravo. A Logic Meta-Programming Framework for Supporting the Refactoring Process. Master's thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, Departement Informatica, Programming Technology Lab, Brussels, Belgium, 2003. In collaboration with Ecole des Mines de Nantes, Nantes, France.
URL: http://prog.vub.ac.be/Publications/2003/vub-prog-ms-03-01.pdf.

[54] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The Story of Moose: an Agile Reengineering Environment. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 1–10, New York, NY, USA, 2005. ACM Press.
ISBN: 1-59593-014-0.
DOI: http://doi.acm.org/10.1145/1081706.1081707.

[55] David Lorge Parnas. Information Distribution Aspects of Design Methodology. *Information Processing*, (71):339–344, 1972.

[56] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
ISSN: 0001-0782.
DOI: 10.1145/361598.361623.

[57] Thomas U. Pimmler and Steven D. Eppinger. Integration Analysis of Product Decompositions. In *ASME Design Theory and Methodology Conference, Minneapolis, MN, USA*, September 1994.
URL: http://www.mit.edu/people/eppinger/pdf/Pimmler_DTM1994.pdf.

[58] Trygve Reenskaug. Thing-Model-View-Editor – an Example from a planningsystem. Technical Note 1979-05-MVC, Xerox PARC, May 1979.
URL: http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf.

[59] Trygve Reenskaug. Models – Views – Controllers. Technical Note 1979-12-MVC, Xerox PARC, December 1979.
URL: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[60] Trygve Reenskaug. The Model-View-Controller (MVC) – Its Past and Present. In *JavaZONE 2003, Oslo*, 2003.
URL: http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf.

[61] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
DOI: 10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.0.CO;2-T.

[62] Neeraj Sangal. Expressing Software Architecture with Inter-module Dependencies. *EclipseZone*, March 2006. Only published online.
URL: http://www.eclipsezone.com/articles/lattix-dsm.

[63] Neeraj Sangal. To Know the Dependencies is to Understand the Architecture. Technical Session TS-6037 held at *JavaOne Conference*, May 2006.
URL: http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-6037&yr=2006.

[64] Neeraj Sangal. To Know the Dependencies is to Understand the Architecture. Presentation held at *Connecticut Java User Group*, January 2007.
URL: http://cooug.org/java/presentations/january2007/index.html.

[65] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 167–176. ACM Press, New York, NY, USA, 2005.
ISBN: 1-59593-031-0.
DOI: 10.1145/1094811.1094824.

[66] Matthias Stevens. DSM+NOV Tool: A DSM+NOV Analysis add-in for Microsoft Excel, 2006.
URL: http://wilma.vub.ac.be/~mstevens/stage-thesis/DSM+NOV.xla.

[67] Donald V. Steward. The Design Structure System. Technical Report 67APE6, General Electric – Atomic Power Equipment Department, San Jose, CA, USA, September 1967. Internal report.

[68] Donald V. Steward. The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, August 1981.
URL: http://www.dsmweb.org/Abstracts/steward.htm.

[69] Maximilian Stoerzer and Juergen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656. IEEE Computer Society Press, Los Alamitos, CA, USA, September 2005.
ISBN: 0-7695-2368-4, ISSN: 1063-6773.
DOI: 10.1109/ICSM.2005.99.

[70] Kevin J. Sullivan. Software Design: The Options Approach. In *Joint proceedings of the second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on*

SIGSOFT '96 Workshops, pages 15–18. ACM Press, New York, NY, USA, 1996.
ISBN: 0-89791-867-3.
DOI: 10.1145/243327.243338.

[71] Kevin J. Sullivan, Prasad Chalasani, Somesh Jha, and Vibha Sazawal. Software Design as an Investment Activity: A Real Options Perspective. In Lenos Trigeorgis, editor, *Real Options and Business Strategy: Applications to Decision Making*, chapter 10, pages 215–262. Risk Books (Incisive Media), London, UK, December 1999.
ISBN: 1-89933-247-2, 978-1-899-33247-2.
URL: http://db.riskwaters.com/public/showPage.html?page=book_page&tempPageName=154032.

[72] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The Structure and Value of Modularity in Software Design. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, September 2001, Vienna, Austria*, pages 99–108. ACM Press, New York, NY, USA, September 2001.
ISBN: 1-58113-390-1.
DOI: 10.1145/503209.503224.

[73] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175. ACM Press, New York, NY, USA, April 2005.
ISBN: 1-59593-014-0.
DOI: 10.1145/1081706.1081734.

[74] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering (May 16-22 1999, Los Angeles, CA, USA)*, pages 107–119. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.
ISBN: 1-58113-074-0.
DOI: 10.1109/ICSE.1999.841000.

[75] Klaas van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan. AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation. Technical Report AOSD-Europe Deliverable D9, AOSD-Europe-UT-01, Universiteit Twente, May 2005.
URL: http://www.aosd-europe.net/deliverables/d9.pdf.

[76] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, Departement Informatica, Programming Technology Lab, Brussels, Belgium, January 2001.
URL: http://prog.vub.ac.be/Publications/2001/vub-prog-phd-01-01.pdf.

[77] Roel Wuyts. StarBrowser, 2003.
URL: http://homepages.ulb.ac.be/~rowuyts/StarBrowser.

[78] Roel Wuyts. RoelTyper, 2005. A fast type reconstructor for Smalltalk.
URL: http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper.

[79] Roel Wuyts, Johan Brichau, Kris Gybels, et al. SOUL, the Smalltalk Open Unification Language, 2001.
URL: http://prog.vub.ac.be/SOUL.

[80] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages, Systems & Structures*, 30(1-2):63–77, February 2004.
DOI: 10.1016/j.cl.2003.08.003.

[81] Ali A. Yassine. An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method. *Quaderni di Management (Italian Management Review)*, (9), 2004.
URL: http://www.iese.uiuc.edu/pdlab/Papers/DSM-Tutorial.pdf.

[82] Ali A. Yassine, Donald R. Falkenburg, and Ken Chelst. Engineering design management: an information structure approach. *International Journal of Production Research*, 37(13):2957–2975, 1999.
ISSN: 0020-7543.
DOI: 10.1080/002075499190374.

[83] AspectJ Quick Reference.
URL: http://www.eclipse.org/aspectj/doc/released/quick.pdf.

[84] AspectJ 5 Quick Reference.
URL: http://www.eclipse.org/aspectj/doc/released/quick5.pdf.

[85] AspectJ Development Tools (AJDT).
URL: http://www.eclipse.org/ajdt.

[86] The AspectJ^{TM} Programming Guide.
URL: http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[87] Apache Ant.
URL: http://ant.apache.org.

[88] The AspectJ Project.
URL: http://www.eclipse.org/aspectj.

[89] The International DSM Conference.
URL: http://www.dsm-conference.org.

[90] DSM Tutorial. DSM Website.
URL: http://www.dsmweb.org/index.php?option=com_content&task=view&id=32&Itemid=30.

[91] The Design Structure Matrix Web Site (DSM Community Site).
URL: http://www.dsmweb.org.

[92] Eclipse, an open-source software development IDE.
URL: http://www.eclipse.org.

[93] Japan, Java Package Analyser.
URL: http://japan.sourceforge.net.

[94] Java programming language.
URL: http://java.sun.com.

[95] Lattix LDM.
URL: http://www.lattix.com/products/LDM.php.

[96] Microsoft Excel.
URL: http://office.microsoft.com/excel.

[97] Microsoft Visual Studio.
URL: http://msdn.microsoft.com/vstudio.

[98] Microsoft .NET Framework.
URL: http://msdn.microsoft.com/netframework.

[99] NDepend, a tool to control the complexity, quality and evolution of .NET code.
URL: http://www.ndepend.com.

[100] SourceForge.net.
URL: http://sourceforge.net.

[101] The Sourcerer Project.
URL: http://sourcerer.ics.uci.edu.

[102] Sun Java 2 Platform Enterprise Edition (J2EE).
URL: http://java.sun.com/javaee.

[103] Unified Modeling Language^{TM}.
URL: http://www.uml.org.

[104] Visual Basic for Applications (VBA).
URL: http://msdn.microsoft.com/isv/technology/vba.

[105] Cincom Systems VisualWorks.
URL: http://www.cincomsmalltalk.com/userblogs/cincom/blogView?content=vwfactsheet.

[106] VisualWorks Trippy Inspector.
URL: http://www.cincomsmalltalk.com/CincomSmalltalkWiki/Trippy+(new+inspector)+Walkthrough.

[107] Foxes Team, XNUMBERS. A library for Microsoft Excel that adds multi precision floating point computing and numerical methods.
URL: http://digilander.libero.it/foxes.