Analysing and Managing Software Dependencies with a Dependency Structure Matrix Tool

University of Oulu
Department of Information
Processing Science
Master's Thesis
Kaisa Kittilä
September 23, 2008

Abstract

Managing software dependencies is a challenging task. None the less, having a healthy dependency structure will prove quite useful throughout the software's life cycle. The understandability of a software product increases when it has distinguishable modules with clear responsibilities and a minimal amount of references between these modules. The combination of high understandability and low coupling between modules significantly eases the development and maintenance of the software.

The extraction of software dependencies by looking at the source code of any nontrivial software program is quite difficult. Software developers can benefit from using automated source code analysis tools that visualise the dependency structure of the software program. These tools can be used in both the development of new software and in the maintenance of existing software.

This research aims at creating a dependency analysis and management tool that represents dependencies in a dependency structure matrix (DSM) form. The research is done using the conceptual-analytical approach and design science. The created artefact is evaluated by comparing its objectives to the actual implementation, and by comparing the artefact to Lattix LDM, a similar commercial tool. Also, the end users' impressions of the tool are collected with a web questionnaire.

The scientific contribution of this research lies in providing an extensive review of dependencies, DSMs, and their use in commercial dependency analysis tools. This research area has not received large amounts of attention in the past, and most of the related research is concerned with doing something specific *with* a DSM tool instead of concentrating on the tool *itself*.

The effects of this research on practice are undeniable: to the author's knowledge, the DSM tool developed during this research is the first open source DSM tool. An even more concrete evidence of success is that the tool is already being used by professional software developers in their work.

Keywords
Dependencies, dependency analysis, dependency management, Dependency Structure Matrix (DSM), dependency analysis tools

Foreword

During my research, I have received help and support from various people, and I would like to sincerely thank you all. This thesis, and its concrete result, the dtangler tool, could not have been created without your assistance and involvement.

Firstly, I would like to thank my employer, Ari Pikkarainen at Sysart, for offering me the chance to work in this interesting project. Naturally, praise is in order to the whole dtangler development team: product owner Daniel Wellner, Scrum master Pertti Lehtisaari, and everyone else who participated and contributed to the development of dtangler. Thanks to Daniel also for reading and giving feedback on my thesis.

I would also like to express my gratitude to my supervisor, Ph. D. Lasse Harjumaa, and my opponent, Ph. D. Ilkka Tervonen, for their insightful comments on my work.

Finally, I would like to thank my parents for their support throughout all my years of study, my fiancé Antti for everything, and my cat Aatu for purring diligently on my lap through most of the time I spent writing this thesis.

Haukipudas, September 13, 2008

Kaisa Kittilä

Contents

1. Introduction

Changes are an essential part of software life cycle. In fact, it is likely that as long as a software product is in use, it will be modified. Uncontrolled changes to software should be avoided. Instead, when altering a software product, it is important to understand the current software and analyze the consequences of this particular change. (Bohner, 2002, p. 264.) This view outlines the importance of thoroughly understanding the software and any design rationale behind it prior to altering the software.

Dependencies between software modules play a substantial role in the ease of making changes to a software product. If the modules are highly dependent on each other, the modification of one module will likely propagate changes to other modules. The terms commonly used in this context are *coupling* and *cohesion*. Modules are highly coupled when there are a lot of connections between them. A connection is simply a reference from one module to another. While coupling measures inter-module dependencies, cohesion represents relationships inside a single module. If the elements of a module constitute a rational, coherent entity, they are most likely closely related and should be dependent on each other. In conclusion, low coupling and high cohesion is sought in order to simplify the structure of a software product. (Stevens, Myers, & Constantine, 1974, p. 116-117, 121.)

The idea of minimizing dependencies between various parts of a software product and creating logical, meaningful, and separate modules is well recognized. The problem lies in how to verify that these design principles have been followed (Huynh & Cai, 2007, p. 1). This becomes increasingly difficult when dealing with larger programs, especially with legacy systems. A legacy system is a large, old software program which is still in active use because replacing it with a new system would cost more than maintaining the old system. The design documentation of legacy software is either out of date or nonexistent. (Bennett, 1995, p. 19-20.). To gain a solid understanding of such a program's source code manually can be very difficult, or even impossible. In situations like this, source code analysis tools, such as DSM (*Design Structure Matrix* or *Dependency Structure Matrix*) tools, can be a big help in visualizing and grasping the program structure.

Even though software changes cannot be prevented, they can be insulated by controlling the dependency structure of the software. The importance of restricting changes to small areas increases as the software grows and advances in its life cycle. The cost of making changes to software is significantly higher in the latter phases of software development, such as testing and maintenance (Boehm & Papaccio, 1988). Performing software changes can take up half of the whole development effort (Boehm & Basili, 2001, p. 135), and also prolong the whole software development project (Gopal, Mukhopadhyay, & Krishnan, 2002, p. 198). Thus, decreasing the amount of rework by improving the dependency structure can yield significant results.

1.1 Research Problem

The research problem of this thesis is:

- How can software dependency analysis with DSMs be automated?

This research problem can be further divided into sub-problems:

1. What are software dependencies and how can they be analysed?
2. What are DSMs and how are they produced?
3. What are the requirements for a DSM tool?
4. What are the benefits and problems of DSM tools?

The first two research sub-problems will be answered with a conceptual-analytical study of dependencies and DSMs. The requirements for a DSM tool can be partially derived from previous literature, and partially by analysing existing DSM tools. The fourth sub-problem can be answered by examining previous research, and by analysing the features of existing DSM tools and the tool created in this research.

1.2 Research Methods

The progress of the research is presented in Figure 1. The analysis phase consists of the literature research done with the conceptual-analytical approach, and the examination of existing tools. The construction and evaluation phase represent the design science approach with which a dependency analysis tool is created and assessed.

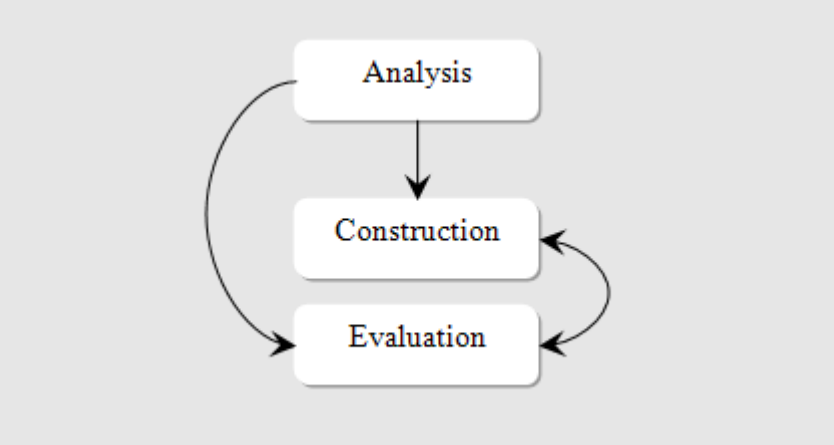


Figure 1. The three main activities of this research.

The arrows in the diagram represent relations between the activities. The analysis phase provides information to the construction and evaluation of the tool. Construction and evaluation are closely linked because the tool is evaluated throughout its construction. The evaluation can drive the construction of the tool in a certain direction. The construction of the tool also has an effect on how it is to be evaluated.

The three activities overlap temporally. The construction of the tool is done concurrently with the literature research. Evaluation is partly done during the construction phase and partly after the construction is finished.

As mentioned, the research is done using the conceptual-analytical approach and design science. Their appliance to this particular research is described in more detail below.

1.2.1 Conceptual-Analytical Approach

The conceptual-analytical approach (Järvinen & Järvinen, 2004, p. 10) is used in identifying and analyzing the essential concepts of the problem area. The information

search is mostly done using Google Scholar and central databases of information processing science, for example, ACM (*Association for Computing Machinery*) and IEEE (*Institute of Electrical and Electronic Engineers*). Various combinations of the following keywords are used in the search:

- software dependencies
- DSM OR “Dependency Structure Matrix” OR “Design Structure Matrix”
- dependency (“management” OR “analysis”)
- tools OR “tool support”

New articles are also discovered from the bibliographies of relevant publications, and by searching who references a particular relevant publication.

The majority of the referenced material is to be selected from journal articles and conference proceedings. Magazine articles, books, and web pages are considered to be scientifically less reliable sources. They are used only if they provide particularly important viewpoints to take under analysis. On the other hand, DSMs in software have not received a vast amount of interest in the form of publications. This has to be taken into account when assessing whether or not a publication should be included in the research.

1.2.2 Design Science

Design science is concerned with the construction and evaluation of an IT (*Information Technology*) artefact (Hevner, March, Park, & Ram, 2004, p. 77). This research uses design science to construct and evaluate a DSM tool. Hevner et al. (2004, p. 83) list guidelines that design science research should follow. Table 1 presents how these guidelines are met in this study.

Table 1. Applying the design science guidelines (Hevner et al., 2004, p. 83) to this research.

Guideline name and description	Description of meeting the guideline in this research
Guideline 1: Design as an Artifact Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.	The produced artefact is an instantiation: a software program.
Guideline 2: Problem Relevance The objective of design-science research is to develop technology-based solutions to important and relevant business problems.	Software dependencies play an essential role in software development. Having a good dependency structure increases software understandability, testability, and eases its evolution. Researching a solution for this problem by constructing a tool to manage software dependencies is thus quite relevant.
Guideline 3: Design Evaluation The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.	Because the development process utilises of TDD (*Test-Driven Development*), the artefact will be constantly evaluated throughout its construction. When completed, the artefact will be evaluated by comparing its features and analysis results to those of other similar tools. Also, the artefact will be evaluated against any requirements identified with research problem three (see Chapter 1.1). The end users' experiences with the tool are evaluated with a web questionnaire.
Guideline 4: Research Contributions Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.	In addition to constructing a DSM tool, which in itself has been quite rare in related research, the tool will be open sourced. To the author's knowledge, no other open source DSM tools exist.
Guideline 5: Research Rigor Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.	The construction of the artefact follows TDD and the Scrum process. The evaluation is comparative: the artefact is compared to a similar product and its own requirements.
Guideline 6: Design as a Search Process The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.	Due to the Scrum process, the construction of the artefact is done in iterations. Each iteration results in a working software, and is one step closer to the finding the solution for automated dependency analysis and management with DSMs.
Guideline 7: Communication of Research Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.	The technology oriented audience will benefit most from the description of the implementation process. The open source code of the artefact can also be studied and applied to other projects. For management oriented audiences, the need for this kind of tool is explained. The tool and its features are presented and evaluated. Also, because the tool is freely downloadable, it can be experimented with in the reader's organisation.

In summary, the research will be performed in iterative steps, and the goal is to produce a dependency analysis and management tool that meets its requirements. The biggest contributions of this research are to create an open source dependency analysis tool and highlight specific issues related to its development. Also, reviewing literature and

existing tools will provide a good amount of background information about dependencies in object-oriented software and their management with tool support.

1.3 Related Research

Dependencies, and their importance to software development, have been studied for decades. Parnas (1979) pointed out the problems of having uncontrolled dependencies between software modules and introduced the concept of information hiding. Stevens, Myers, and Constantine (1974) studied the nature of dependencies in detail and presented coupling and cohesion as a means to characterise dependencies. Various software metrics have been designed to identify and analyse coupling and cohesion more formally (Bieman & Kang, 1995; Briand, Devanbu, & Melo, 1997; Chidamber & Kemerer, 1994; Dhama, 1995; Eder, Kappel, & Schrefl, 1992; Hitz & Montazeri, 1995).

Dependencies and their presentation have been studied by various researchers. Cox, Delugach, and Skipper (2001) have created a more formal way to model dependencies accurately with conceptual graphs. Guo (2002) had the same goal of formalizing dependency modelling but he incorporated category theory to model dependencies. Jackson's (2004) dependency model is not as formal as the previous two; it can be thought of more as a textual definition to further define the meaning of a dependency. Vieira and Richardson (2002) take a more practical approach by defining a method for analysing software component dependencies.

The term DSM was introduced by Steward (1981). Eppinger (1991) has researched how DSMs can be utilised in the analysis and management of concurrent tasks. Virtually anything that requires planning and resourcing of large and concurrent tasks can benefit from DSM modelling. In fact, DSMs have been used in a variety of fields, for example aerospace (Rogers, Korte, & Bilardo, 2006) and building construction (Austin, Baldwin, & Newton, 1996; Oloufa, Hosni, Fayez, & Axelsson, 2004).

In software, DSMs are used to explore the dependencies of a software architecture (Huynh & Cai, 2007; MacCormack, Rusnak, & Baldwin, 2006; Sangal, Jordan, Sinha, & Jackson, 2005), or to model dependencies on design level (Cai, 2006; Lopes & Bajracharya, 2005). Matos, Duarte, Cardim, and Borba (2007) have analysed aspect-oriented product lines with DSMs.

Baldwin and Clark (2000) use DSMs to visualise the structure of software design in their design rule theory. Design rules capture the fundamental part of a software program; they are decisions that are not likely to change during the program's life cycle. In essence, they make up the interfaces of software modules. DSMs have been applied in the context of Baldwin and Clark's design rule theory by Lopes and Bajracharya (2005), Sullivan, Griswold, Cai, and Hallen (2001), and LaMantia, Cai, MacCormak, and Rusnak (2008).

This thesis differs from the majority of related research in that it is concerned with the construction of a DSM tool instead of the analysis of a specific software or design with an existing tool. Also, a large amount of research on DSMs is done on design level analysis of dependencies, especially when DSMs are used in conjunction with Baldwin and Clark's (2000) design rule theory. The DSM tool implemented in this research analyses dependencies on implementation level.

1.4 Scope of the Research

This thesis focuses on dependencies mainly in object-oriented software. The constructed artefact concentrates even further on Java dependency analysis.

While the artefact is naturally evaluated as a part of this study, it is acknowledged that further evaluation, preferably done by observing the use of the tool in real-life software projects, is needed to fully assess the effect the tool has on the development process and on the end product. The main contribution of this research lies in the construction of a complete dependency analysis tool; its detailed evaluation would yield enough material for a whole new research.

1.5 Organisation of the Thesis

Chapter 2 identifies and analyses the general concepts that relate to software dependencies and dependency management. It also elaborates on how DSMs can be used in dependency modelling.

Chapter 3 presents methods for formally measuring the different aspects of dependencies, including coupling, cohesion, and stability.

In Chapter 4, various tools for dependency analysis with DSMs are presented.

Chapter 5 sets the requirements for the dependency analysis tool that was created during this research. The implementation of the tool is also described.

The artefact is evaluated in Chapter 6. First, the tool is evaluated in light of the requirements that were defined in Chapter 5. The features of the constructed artefact are also compared to those of an existing, similar product. Also, the outcome of a dependency analysis from a test program is compared between the artefact and the existing product. Finally, the tool is evaluated by analysing the results of an end user survey.

Chapter 7 concludes the research and answers explicitly to the research questions set in this introductory chapter.

2. Software Dependencies

The quality of software dependencies has a great impact on software understandability, reusability, maintainability, and testability (de Souza, 2005, p. 21-22). In this chapter, the importance of achieving and maintaining a healthy software dependency structure is explained in detail.

The definition of software dependencies varies between different research publications, according to the focal point taken in each research. The most general definition is that a dependency is a connection between two items in which changes to one item may cause changes to the other item as well (Wilde, 1990, as cited in de Souza, 2005, p. 18). The different ways to categorise and classify dependencies are discussed further in this chapter. Also, dependency modelling, especially with DSMs, is described.

2.1 Changes Are an Integral Part of Software Development

Software is, by its nature, intended to change throughout its life cycle. Without proper care, it will deteriorate. This can, for example, be manifested as outdated documentation, or complicated design or code structure. (Pfleeger & Bohner, 1990, p. 320.) In addition to the gradual growth of complexity and reduction of maintainability, there is a risk that defects are introduced into the system whenever new features are implemented, or existing features are modified (Ohlsson, von Mayrhauser, McGuire, & Wohlin, 1999, p.69).

Dependencies between software modules play a major role in software maintainability. This is due to the fact that changes propagate through dependencies. When you modify one software module, it is possible that other modules dependent on this module will also need to be changed. (Ohlsson et al., 1999, p. 71.) The likelihood of a change affecting other modules increases with the number of dependencies a module has. As software systems grow more complex and large, the significance of dependency management grows as well (Guo, 2002).

Legacy software is a concrete example of how software inevitably degrades as time goes by. With legacy software, the original idea and design have been lost during years of maintenance. The original developers might have moved to work elsewhere, or retired completely. The only up-to-date documentation that remains is the source code itself. (Bennett, 1995; Chen & Rajich, 2001.)

What can be done to reduce the negative impact of software changes? The first step is to identify the modules which are most likely to deteriorate. If historical data from previous software versions is available, it can be used to predict the stability of each module. This way, developers can focus their efforts on the identified problem areas. (Grosser, Sahraoui, & Valtchev, 2003.)

Identifying less stable modules by relying on historical data is naturally more useful with older software products, or with products that release new versions frequently. Stability prediction has been applied by Ohlsson et al. (1999) to a large C program with multiple successive releases. Grosser et al. (2003) have analysed and predicted class-

level design stability between different versions of the Java API (*Application Programming Interface*).

Impact analysis is another approach to managing software changes. In impact analysis, the effect of introducing a change is evaluated before actually implementing anything. This makes it possible to weigh the positive effects of the proposed change against any negative side effects, and decide on the best course of action for the current situation. Impact analysis can also be used to select the most beneficial solution from a group of candidate solutions. (Pfleeger & Bohner, 1990, p. 320.)

Pfleeger and Bohner (1990) introduce a complete process for software maintenance tasks. More generally, this process can be used whenever making changes to software, regardless of the development phase. The goal is to make software changes more controlled. The process combines the use of different software metrics and impact analysis, and includes the examination of system dependencies as a central part of the impact analysis.

2.2 Basic Terms and Definitions

In general, dependencies can be direct, transitive, or cyclic. Direct dependencies exist between two items, *dependent* and *dependee*. For example, in Figure 2, the dependency between A and B is direct; A has the role of the dependent, B the role of dependee. The same applies to the dependency between B and C. (Jungmayr, 2002, p. 1.) Transitive and cyclic dependencies are discussed in more detail in the following sub-chapters.

2.2.1 Transitive Dependencies

In addition to two direct dependencies, Figure 2 shows a transitive dependency between A and C. A transitive dependency occurs when two items are linked via one or more intermediary nodes (Jungmayr, 2002, p. 1). Transitive dependencies are not as significant as direct dependencies. If a software system is designed properly, and follows basic object oriented principles, such as encapsulation and modularity, it is likely that a majority of changes will not propagate through transitive dependencies. (Jackson, 2004, p. 199.) In fact, UML (*Unified Modelling Language*) does not acknowledge transitive dependencies as actual dependencies; if the dependency between two items is not direct, the items are presumed to be mutually independent (Fowler, 2001, p. 103).

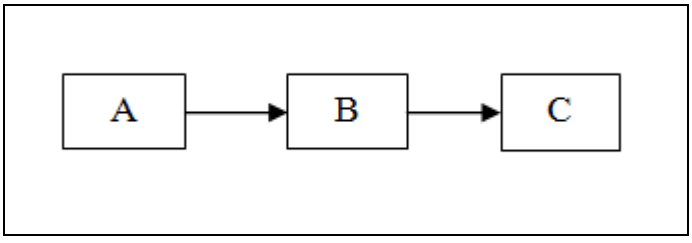


Figure 2. Two direct dependencies, {A, B} and {B, C}, and one transitive dependency, {A, C}.

Even if transitive dependencies aren't as consequential as direct dependencies, they can still have negative effects on software quality. Lakos (1996, p. 3, 14) points out that transitive dependencies can reduce the understandability and reusability of software modules: in order to fully understand a module, you also have to understand all the

modules which it depends on, either directly or transitively. Jungmayr (2002) adds that testing a module with a lot of dependencies becomes harder because all dependee modules, transitive or not, have to be considered in the test.

2.2.2 Cyclic Dependencies

Cyclic dependencies are considered the most harmful dependency type. Their use is discouraged in most situations, especially on higher abstraction levels, such as on Java package level. (Fowler, 2001, p. 2-3; Lopes & Bajracharya, 2005, p. 17; Martin, 1996, p. 6-7; Melton & Tempero, 2006, p. 35-36; Sangal et al., 2005, p. 168.) A cyclic dependency occurs when dependent items create a directed graph that starts and ends in the same node. The simplest cyclic dependency contains two items which depend on each other (see Figure 3). (Melton & Tempero, 2007, p. 395.)

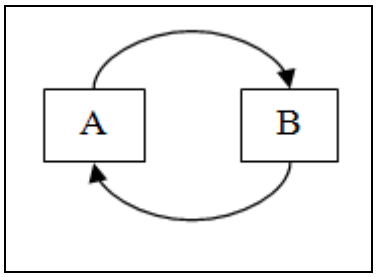


Figure 3. A cyclic dependency between two items.

There are different approaches to defining how dependencies qualify as cyclic dependencies. One definition requires the cycle participants to appear in the dependency path exactly once. Cycles of this type are called simple cycles (Melton & Tempero, 2007, p. 395). SCCs are sub-graphs of directed graphs, in which every node can be reached from any other node in the sub-graph (Gross & Yellen, 2004, as cited in Melton & Tempero, 2007, p. 395). Depending on the definition of a cycle, the graph in Figure 4 contains either two or three cycles. The cycle containing all three items is not a simple cycle because it is not possible to start and end in A by visiting both B and C only once. Still, it is a SCC cycle because all nodes are navigable from all nodes.

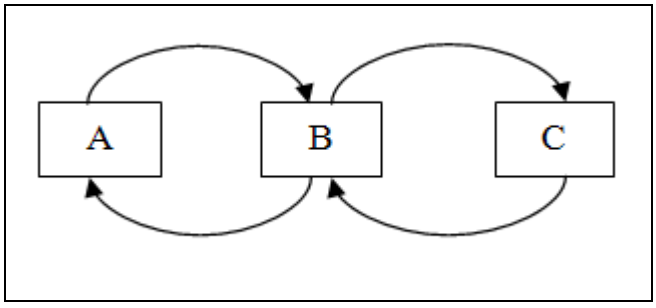


Figure 4. In addition two simple cycles {A, B}, {B, C}, the graph contains one SCC cycle {A, B, C}.

Melton and Tempero (2007) have analysed the amount of cyclic dependencies in numerous Java programs. Their findings are in line with previous observations which state that even though it is inadvisable to have cyclic dependencies, they are quite common in real-life programs. Nearly all of the programs included in the study

contained a lot of long cyclic dependencies. It was also noted that the amount of cyclic dependencies dropped whenever a major refactoring had taken place between software versions. (Melton & Tempero, 2007, p. 408-409.)

One of the reasons for the abundance of cyclic dependencies in software programs is the lack of methods for identifying them. Long cycles involving dozens of modules appear to be quite common in real-life software programs (Melton & Tempero, 2007, p. 408-409). Cycles become harder to identify when the number of modules contributing to a cycle grows.

2.3 Dependency types

A variety of dependency types have been presented and analysed in previous research. To summarise, dependencies can be classified by their abstraction level, by their static or dynamic nature, or by their impact weight (see Table 2).

Table 2. Summary of different dependency types found in literature.

Dependency category	Dependency type	Description	References
Abstraction level of dependency	Design dependencies	Dependencies between different design artefacts, for example, between different software requirements	(Cai, 2006; Huynh & Cai, 2007; Lopes & Bajracharya, 2005; Pfleeger & Bohner, 1990)
	Implementation dependencies	Dependencies between different parts of a software program, for example, between classes or packages	(Chen & Rajich, 2001; Dhama, 1995; Huynh & Cai, 2007; Pfleeger & Bohner, 1990)
	Cross-level dependencies	Traceability between different abstraction levels, for example, from what design document a certain software module is derived from	(Huynh & Cai, 2007; Murphy, Notkin, & Sullivan, 2001; Pfleeger & Bohner, 1990)
Dependency extraction method	Static dependencies	Dependencies that can be identified at compile time	(Jungmayr, 2002; Melton & Tempero, 2007; Tuura, 2003; Xiao & Tzerpos, 2005)
	Dynamic dependencies	Run-time dependencies	(Melton & Tempero, 2007; Tuura, 2003; Xiao & Tzerpos, 2005)
	Knowledge dependencies	A software module has to make assumptions about the implementations of other software modules.	(Chen & Rajich, 2001; Jackson, 2004; Jungmayr, 2002; Tuura, 2003; Yang & Tempero, 2007)
Dependency significance	Interface dependencies	A software module uses another module through the provided interface.	(Hitz & Montazeri, 1995; Melton & Tempero, 2007; Nordberg III, 2001; Stevens et al., 1974)
	Implementation dependencies	A software module is directly dependent on another module's implementations	(Hitz & Montazeri, 1995; Melton & Tempero, 2007; Nordberg III, 2001; Stevens et al., 1974)

The first dependency category is concerned with abstraction levels. Dependencies can be analysed on design or implementation level, or by tracking how artefacts on different levels depend on each other. The main focus in this thesis is on implementation level dependencies. The following two dependency categories, dependency extraction time and dependency significance, apply to implementation level dependencies.

The dependency extraction category classifies dependencies into three groups: static, dynamic, and knowledge. Static dependencies can be extracted from binary files; dynamic dependencies can contain additional dependencies introduced during run-time, for example, dynamic class loading (Tuura, 2003, p. 684-685). Knowledge dependencies are more subtle, and need to be analysed manually. They can also be referred to as conceptual (Chen & Rajich, 2001, p. 230) or logical (Jungmayr, 2002, p. 1) dependencies.

Knowledge dependencies are harder to identify and manage than static and dynamic dependencies. Knowledge dependencies occur when a change to an item affects another item even when they are not directly dependent on each other. Knowledge dependencies are extracted by identifying how software modules are conceptually dependent on each others' implementations and data structures. (Chen & Rajich, 2001, p. 230; Yang & Tempero, 2007.)

For example, a knowledge dependency exists when a class has to make assumptions about the internal structure of another class to be able to perform its own task correctly. Knowledge dependencies are problematic because any errors resulting from false assumptions or protocol breaches can be difficult to track down. For example, when an object doesn't handle a shared variable according to protocol and leaves the variable in a broken state, the resulting error might not manifest itself until the variable is used again at some time in the future. Finding the underlying cause for the failure can prove to be hard. (Chen & Rajich, 2001, p. 230; Tuura, 2003, p. 685.; Yang & Tempero, 2007.)

The third category in Table 2 divides dependencies according to their significance, or impact weight. This categorisation emphasises that some dependencies are more harmful than others. In general, depending on interface is more favourable than depending on implementation. Class A depends on class B's interface if it accesses B's functionality only through its methods. An implementation dependency occurs when class A manipulates B's instance variables directly. (Hitz & Montazeri, 1995, p. 2; Nordberg III, 2001, p. 5.)

Even though interface dependencies are recommended, it should be noted that the quality of the interface naturally has an effect on the significance of the dependency (Melton & Tempero, 2007, p. 395; Stevens et al., 1974, p. 119). This view is also supported by Takahashi and Nakamura (1996), whose research shows that a software program's error density increases as the complexity of its interfaces increase.

Information hiding, a programming principle articulated by Parnas (1979), can be used to reduce the negative effects that dependencies may have. Encapsulating unnecessary detail not only makes it easier to understand a module but also makes it easier to change the actual implementation without affecting other modules. If modules are directly dependent on the implementation, not the interface, changes are likely to propagate in a wider range. Thus, interface dependencies are favoured over implementation dependencies.

2.4 Dependency Modelling with DSMs

Dependencies are usually modelled with different variations of box-and-arrow graphs, such as UML graphs. Boxes represent the components being modelled and arrows show dependencies between components. (Cai, 2006, p. 6.) Modelling dependencies with box-and-arrow graphs is quite intuitive and makes the dependencies easily discernible.

Dependencies between different parts of software can also be presented in a matrix form, as DSMs. When the amount of components in the model grows, box-and-arrow diagrams tend to get too cluttered and hard to understand. DSMs, on the other hand, maintain the same visual representation regardless of their size; you only add more rows and columns. Because of their scalability, DSMs are well suited for dependency modelling in large architectures. (Sangal et al., 2005, s. 168.)

A DSM is a square matrix in which the items being analysed make up the rows and columns. An entry in the matrix indicates that the item on the corresponding column depends on the item on the corresponding row. Dependencies on the diagonal, from upper left to lower right, are not of interest because they would only indicate that an item depends on itself. (Sangal et al., 2005, p. 168; Yassine, 2004, p. 2.)

Figure 5 provides a small example DSM. In this DSM, A depends on B and C; B depends on C. C doesn't depend on neither A nor B because column C is contains no crosses. The example DSM is binary; it only shows whether there is a dependency between two items, all dependencies are considered to be of equal weight. In numerical DSMs, the crosses are replaced with numbers that show the exact dependency weights. (Yassine, 2004, p. 8-9.)

	A	B	C
A	-		
B	X	-	
C	X	X	-

Figure 5. A simple DSM.

Browning (2001, p. 293) sees DSMs fundamentally as a tool that can help develop software more systematically and innovatively. It is easy to see that using DSMs can bring a more methodical approach to software development but how can DSMs make software development more innovative? Browning (2001, p. 293) reasons that in order to make innovative architectural decisions, you first have to understand the existing design thoroughly. DSMs can efficiently increase the developers' understanding of a particular design because they highlight the system structure with emphasis on dependencies between software components.

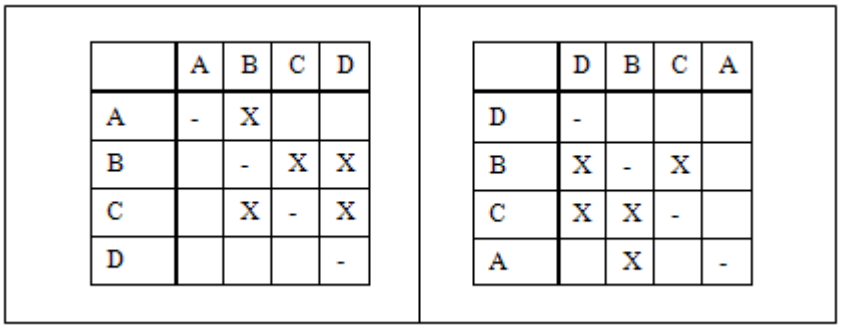
The ordering of row items on the matrix is important because it dictates what the DSM will look like and how it should be interpreted. The row order can be defined with different algorithms, mainly clustering and partitioning algorithms. (Browning, 2001; Sangal et al., 2005.) After the rows have been ordered, any anomalies in the produced DSM point to areas that might require redesign.

Clustering algorithms strive to group interdependent items into clusters and order these clusters so that all dependencies are as near the diagonal as possible. Any dependencies

far from the diagonal or outside the defined clusters indicate possible problem areas. (Browning, 2001, p. 293-294; Sangal et al., 2005, p. 168.)

Partitioning algorithms attempt to order the rows so that all dependencies are below the diagonal. This is achieved by positioning the items that have no *dependents* to the top of the DSM, and by positioning the items that have no *dependees* to the bottom of the DSM. These items and their dependencies are then removed from the partitioning and the remainder of the DSM is processed in a similar fashion. This is repeated until there are no more items left, or until there are no items with zero dependents or dependees left. Any leftover items contain cyclic dependencies. (Sangal et al., 2005, p. 168-169; The Design Structure Matrix Web Site, 2008; Yassine, 2004, p. 6-7.)

The idea behind partitioning is to layer the DSM so that the items only depend on items on lower rows. A properly layered system yields a partitioned DSM with all entries under the diagonal. DSMs of this type are called lower triangular. Any items that remain above the diagonal after partitioning are an indication of cyclic dependencies. (Sangal et al., 2005, p. 168-169; The Design Structure Matrix Web Site, 2008; Yassine, 2004, p. 6-7.)



	A	B	C	D
A	-	X		
B		-	X	X
C		X	-	X
D				-

	D	B	C	A
D	-			
B	X	-	X	
C	X	X	-	
A		X		-

Figure 6. Partitioning a DSM to lower triangular form.

Figure 6 shows an example of DSM partitioning. The DSM on the left is unordered. When partitioning the DSM, it is noted that row D has no entries, and thus, item D isn't providing services to the other items. Likewise, column A is empty, which tells that A isn't dependent on any of the other items. This leads to ordering the rows so that D is the highest and A the lowest row. A and D can then be excluded from the partitioning but the remaining sub-DSM cannot be partitioned further because B and C both have a dependent and a dependee. The resulting partitioned matrix reveals a cyclic dependency: it is impossible to order the rows so that all entries would remain below the diagonal.

2.5 Summary

Managing software dependencies is important because of the fundamental nature of software as something that is guaranteed to change, evolve, and expand over its life cycle. Dealing with software changes becomes less cumbersome if the dependency structure of the software is in good order. Minimising dependencies between software modules helps isolate the impact of changes to a fewer modules.

Software dependencies should be created in a controlled fashion, and also re-evaluated on a timely basis. This can help avoid and root out unwanted dependencies, such as

cyclic dependencies and dependencies on implementation. A clear and logical dependency structure eases further development and maintenance.

A dependency is a relationship between items in which a dependent item requires or uses something that a dependee item provides. Dependencies can be simple direct dependencies with one dependent and one dependee, or transitive dependencies that create a dependency path between multiple items, or cyclic dependencies, in which the dependency path starts and ends with the same item.

Researchers have focused on different aspects of dependencies in the past. Some have studied dependencies from design's viewpoint, some from implementation's. The conformation of a design to its implementation has also received some attention.

The dependency structure of any nontrivial software can quickly become too complex to comprehend just by looking at the source code. Modelling dependencies eases the dependency analysis process. There are various ways to model and examine the dependencies in a software system, ranging from informal box-and-arrow diagrams to UML diagrams and DSMs.

Reading DSMs might require a slight learning curve; box-and-arrow diagrams are more intuitive. Also, transitive dependencies are not easily discernible from DSMs because following the dependency paths requires scanning and jumping between rows and columns. The strength of DSMs lies in their scalability and cycle visualisation. This is a clear benefit when dealing with large architectures. Row ordering techniques, such as DSM partitioning or clustering, can also help in verifying the system architecture and finding the most critical areas for improvement.

3. Measuring Software Dependencies

Formal measuring is needed to analyse different software properties reliably. Measurement theory describes measuring as a well-defined process in which the attributes of real-life entities are assigned with descriptive figures or symbols. Measuring can be used either to predict the outcome of a certain attribute, or to assess the value of a certain attribute. (Fenton, 1994, p. 199.)

When using metrics, a formal model that rules out any possible ambiguity from the measuring needs to be defined. Fenton uses lines of code (LOC) as an example. First, you have to define what exactly is meant by a line of code, for example, whether or not comment lines and empty lines are included in the calculation. (Fenton, 1994, p. 199.)

To model dependencies accurately, it is important to define how dependencies are measured. This chapter examines metrics related to dependency measurement and their variations when applicable.

Coupling and cohesion are probably the best-known metrics when it comes to measuring dependencies. Coupling is used to describe how modules depend on each other, while cohesion describes what the modules' internal dependencies are like (Stevens et al., 1974, p. 116-117, 121). Stability is a dependency metric related to how software changes. It tells what modules are most likely to change by examining the dependencies between the modules. (Martin, 1997.)

3.1 Coupling and Cohesion

Coupling and cohesion are well-known software metrics presented by Stevens, Myers, and Constantine (1974). In short, coupling measures the weight of a dependency between two modules, and cohesion measures dependency weights inside a single module. Low coupling improves software understandability and facilitates making changes to the software. Coupling can be minimised by either reducing inter-module dependencies or by increasing internal dependencies. (Stevens et al., 1974, p. 116-117, 121.)

Even though coupling and cohesion were first introduced in the context of structured design and programming, they are very relevant to object-oriented programming as well. High coupling makes the program harder to understand, develop, and test. (Hitz & Montazeri, 1995, p. 1-2.)

Stevens, Myers, and Constantine (1974) don't give any formal methods for measuring coupling and cohesion. Instead, they provide instructions on how to informally analyse dependencies and identify possible problem areas. Other researchers, including Chidamber and Kemerer (1991; 1994), and Hitz and Montazeri (1995), have elaborated the definition of coupling and cohesion, and attempted to create formal models for coupling and cohesion. These models will be discussed in the following sub-chapters after the principles of coupling and cohesion have been presented.

3.1.1 Coupling: Inter-Module Dependencies

Stevens, Myers, and Constantine (1974, p. 116-118) measure coupling between two modules by analysing the dependency weight between the modules. The dependency weight is a sum of three factors: interface complexity, connection type, and communication type (see Table 3). The researchers' definition for the term interface differs from what it is in object-oriented design. Here, two modules interface if they both use the same common environment, such as a global variable or control block.

Table 3. Factors affecting coupling (Stevens et al., 1974, p. 116-121).

Factor	Description
Interface complexity	How simple and obvious the modules' use of a common environment is?
Connection type	Does the call refer to implementation or to interface?
Communication type	What is the purpose of the communication between these modules? For example, data passing or control flow.

Interface complexity, as defined by Stevens, Myers, and Constantine, cannot be directly applied to object-oriented software because the use of global variables and similar programming practices are discouraged in the object-oriented paradigm to start with. Regardless, the basic idea of more complex connections accounting for more coupling does appear to be applicable in object-oriented software as well.

Another way of viewing interface complexity is by comparing this definition of interface to knowledge dependencies. As recalled from Chapter 2.3, knowledge dependencies are dependencies in which a change to item A can affect item B even though there is no direct dependency between A and B. This could happen, for example, when both A and B access and use the same data provided by item C. When applied to interface complexity, item C is the common environment that both A and B use, hence A and B interface. The fundamental principle is the same in both cases. Thus, interface complexity could be seen as one form of knowledge dependencies.

Weighing dependencies according to communication type, data or control flow, is not very applicable to object-oriented software because the fundamental idea in object-orientation is to keep data and its functionality together.

Out of these three coupling factors, the connection type is most relevant to dependencies in object-oriented software. Basically, it defines whether a module is depending on another module's implementation or interface. This same idea of dependency significance was also used in Chapter 2.3 as a means of dependency categorisation.

Chidamber and Kemerer (1991, p. 202-203) measure coupling with a metric called CBO (*Coupling Between Object classes*). CBO simply calculates how many classes a class references. Dependencies caused by inheritance are ignored. CBO is only concerned with the existence of a dependency, no matter what the weight of that dependency might be. CBO is a metric for analysing how coupled a single class is in junction with all other classes, and not a metric for analysing how coupled any two classes are. This strays away from the original definition of coupling by Stevens, Myers, and Constantine (1974, p. 117), which states that coupling measures the strength of a single dependency.

Hitz' and Montazeri's (1995, p. 4-7) coupling metric resembles more closely the original definition of coupling: it measures the strength of a dependency between two items by analysing by analysing the exact cause and type of the dependency.

Hitz and Montazeri (1995, p. 4-7) divide coupling into two groups: object coupling and class coupling. Object coupling is a more elaborated version of CBO. In addition to disregarding inheritance dependencies, as CBO does, it also ignores local attributes of a method and composite attributes. The other coupling group, class coupling, includes these dependencies into the dependency measurement. Measuring class level coupling is mainly useful whenever changes are made to the system, and object level coupling is relevant to run-time activities, such as testing.

Dhama (1995) has defined a model for measuring coupling in procedural programs. At least one portion of the model, the analysis of environmental coupling, can be applied to object-oriented programs as well. Environmental coupling is calculated by summing up references to and from a module. (Dhama, 1995, p. 71-72.) Mixing up incoming and outgoing references like this hides important characteristics of coupling; you cannot tell whether a certain module is highly coupled because it provides a lot of services to other modules, or because it uses a large number of services from other modules, or both. Again, similar to CBO, this metric models items, not their dependencies.

3.1.2 Cohesion: Intra-Module Dependencies

Stevens, Myers, and Constantine (1974, p. 121) refer to cohesion simply as a means to reduce coupling. This could suggest that the authors deem coupling more significant to software quality than cohesion. Hitz and Montazeri (1995, p. 2) remind that a software system with low coupling is not necessarily an example of good design. If taken to an extreme, a system as such would consist of only one large abstraction of everything. Evaluating the system's cohesion would most likely reveal that some degree of modularity is required in order to achieve modules with a single, well-defined purpose.

Stevens, Myers, and Constantine (1974, p. 121) measure cohesion by examining the purpose of each dependency between the elements of a module. These various dependency types are categorised into six groups. The groups are summarised in Table 4 in descending order. The first group accounts for the weakest cohesion, and the last for the strongest cohesion.

Table 4. Relationship types used in measuring cohesion (Stevens et al., 1974, p. 121-125).

Relationship type	**Description**
Coincidental	No conceptual relationship between the elements
Logical	The elements are of the same general type and handle everything related to that type. For example, a module for all input and output
Temporal	The elements are connected logically and accessed in the same time period
Communicational	The elements manipulate the same data
Sequential	The elements are accessed one after the other
Functional	The elements perform different aspects of a single function

Chidamber and Kemerer (1991; 1994) have defined LCOM (*Lack of COhesion in Methods)* as a metric for measuring cohesion. LCOM is measured from how the methods in a class use its instance variables (Chidamber & Kemerer, 1994, p. 488)[1]:

> Consider a Class C_1 with *n* methods M_1, M_2, …, M_n. Let $\{I_j\}$ = set of instance variables used by method M_i.
>
> There are *n* such sets$\{I_1\}$, ... , $\{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$. If all *n* sets $\{I_1\}$, ... , $\{I_n\}$ are $\varnothing$ then let $P = \varnothing$.
>
> $$LCOM = |P| - |Q|, \text{ if } |P| > |Q|$$
> $$= 0 \text{ otherwise.}$$

LCOM subtracts the number of method pairs that have no common instance variables from the number of method pairs that have at least one common instance variable. Thus, if the methods use a lot of the same instance variables, LCOM is low and cohesion is high. (Chidamber & Kemerer, 1994, p. 488-490.)

Intuitively, LCOM would seem to be a good method for measuring cohesion. It makes sense that the methods in a highly cohesive class should use the same instance variables instead of being independent, each manipulating a separate instance variable. However, Hitz and Montazeri (1995, p. 8-9) argue that this metric does not measure cohesion correctly. They point out that the use of getter and setter methods, considered a fundamental principle of object-oriented programming, would constitute to an unnecessarily high LCOM. A similar issue arises when a method doesn't use instance variables at all, but delegates its work to other, simpler methods of the class. They also argue that a cohesion metric shouldn't be dependent on the number of methods in a class.

Li and Henry's (1993, p. 55) version of the LCOM metric calculates the number of disjoint method sets. Methods that use at least one common instance variable are put in the same set. If there is a small amount of methods with common instance variables, there are more disjoint method sets and thus, LCOM is higher. Hitz and Montazeri (1995, p. 8-9) have modified this LCOM metric further by also accepting methods that call each other to the same method set.

3.2 Stability

While coupling measures dependencies between items, stability is used as a measure to describe single items. Stability is a relevant measure when dependencies are concerned because an item's stability affects how significant its dependencies are. Hitz and Montazeri (1995, p. 4), for example, use stability evaluation as a part of measuring coupling. Weights are applied to different coupling types ranging from accessing a stable class through its interface to accessing the implementation of an unstable class directly. The researchers don't define any formal way of measuring the stability of a class. Stability is approximated by predicting the probability of making a change to a class.

---

[1] The referenced text contains a slight typing error in the formula: Instead of I_j, I_i should represent the set of instance variables used by method M_i. This error was not present in the previous work by the authors (Chidamber & Kemerer, 1991, p. 203) in presenting the preliminary LCOM metric.

Martin (1997, p. 8) defines stability as ease-of-change: the harder it is to make changes to an item, the more stable that item is. If an item is hard to change, it is likely that it will remain unchanged. While ease-of-change and probability-of-change define the basic idea of stability, they are not formal models. Martin (1997, p. 9) proposes a formula that calculates the proportion of dependencies going outwards from an item in comparison to dependencies coming inwards to an item:

$$\frac{Ce}{Ca + Ce} \quad (1)$$

Ce measures *efferent* couplings, that is, on how many items the current item depends on. Ca stands for *afferent* couplings. It describes how many items are dependent on the current item. The instability value is in the range [0,1]: 0 means maximum stability, 1 maximum instability. (Martin, 1997, p. 9.) For example, if a Java package depends on two classes outside its own package but no class from any other package depends on it, Ce is 2 and Ca is 0. This results in instability value 1, maximum instability. A package with an instability value 0 would be completely stable, as it would not be dependent on any other package.

Martin (1994, p.1, 4, 6) uses stability as a measure for dependency health. The dependency between two items is unhealthy if a more stable item depends on a less stable item. This could propagate changes from the less stable item to the more stable item, and, in turn, to any items that are dependent on the stable item. To avoid this kind of large scale change propagation, an item should only depend on an item more stable than itself.

3.3 Summary

While coupling and cohesion both measure dependencies, they have a different outlook.

Coupling is used to describe the dependencies between the different parts of a software program. In short, low coupling means having a small number of justified dependencies between modules, and, as such, is an indication of a healthier dependency structure.

Stability is used in conjunction with coupling to measure the weight of a particular dependency: depending on an instable item increases the dependency weight and makes the items more tightly coupled.

While coupling is used to measure the software as a whole and see how the different modules relate to each other, cohesion looks inside single modules and measures their internal dependencies. High cohesion implies that the insides of a module create a consistent and logical entity with well-defined responsibilities.

The term modularity is closely linked to cohesion. A modular software design consists of highly cohesive modules. This, in turn, reduces coupling; dependencies between modules can be kept at a minimum because responsibilities are not spread randomly between different modules.

4. Tool Support for Dependency Analysis

While dependency models can be created manually, automating the dependency extraction process can provide results faster and more accurately. Automated tools can also help make dependency analysis a solid part of the software development process. For example, some tools can be used to define project specific design rules (Sangal et al., 2005, s. 167-168). The implementation's compliance to these rules can then be checked whenever changes have been made to the software.

The purpose of this chapter is to present and compare different dependency analysis tools. The emphasis is on commercial tools with DSM modelling capabilities.

4.1 Research Background on Dependency Analysis Tools

A lot of research on software analysis tools focuses on the maintenance phase of the software life cycle. This can be seen as an indication of the present situation: formal dependency analysis is not thought of during implementation. It would seem that automated tools enter the picture when maintenance becomes too difficult or impossible without the use of such tools. On the other hand, perhaps the amount of interest in this particular area is proportional to the challenges it provides; software maintenance clearly is in need of impact analysis techniques and software structure visualisation.

Chen and Rajlich (2001) have developed a tool, RIPPLES, for impact analysis. With the developer's help, the tool constructs a graph that shows which components of the software would be affected by the change and how these components depend on each other. Visualising the outcome of a change helps verify whether the planned implementation is right for the situation.

Mancoridis, Mitchell, Chen, and Gansner (1999) have worked on Bunch, a tool that creates clusters by analysing source code. The tool is useful whenever up-to-date high-level software designs are not available.

Melton and Tempero's (2006) tool, Jepends, identifies cyclic dependencies from source code. The tool's main purpose is to point the user to those areas in the source code that need refactoring the most.

Cai (2006) created a tool that generates design-level DSMs. The generated DSMs model relationships between different design decisions. They can be used to evaluate the pros and cons of proposed candidate solutions. A more recent research (Huynh & Cai, 2007) expands this work by creating a genetic algorithm that analyses how well design level and implementation level DSMs correspond to each other. A similar tool for comparing design and implementation was developed by Murphy, Notkin, and Sullivan (2001). This tool doesn't generate DSMs; it models dependencies with box-and-line diagrams and textual notations.

Rusnak (2005) created a DSM tool, DSAS (*Design Structure Analysis System*) that creates DSMs from function call dependencies. The tool can analyse the likelihood of changes propagating through dependencies, and cluster the modules to improve the dependency structure.

4.2 Commercial DSM Tools

Some of the DSM tools available are not aimed for software dependency analysis. These tools can be used to, for example, model processes to discover the optimal process task sequence. This section focuses specifically on DSM tools for software dependency analysis.

The core feature of any automated software DSM tool is to generate a matrix representation of inter-module dependencies from source code. Advanced DSM tools can also be used to create dependency rules that define which dependencies are allowed and which are forbidden. DSM tools can also be used in refactoring: some tools can suggest a better architectural structure in order to eliminate unwanted dependencies. (Sangal et al., 2005, s. 167-168.)

The tools Lattix LDM (http://www.lattix.com/products/LDM.php), IntelliJ IDEA (http://www.jetbrains.com/idea/), and Structure101 (http://www.structure101.com/) are presented in this chapter. LDM and Structure101 come in different versions for different dependency inputs. In this presentation, the Java versions of these tools are used.

4.2.1 Lattix LDM

Sangal, Jordan, Sinha, and Jackson (2005) have experimented on using Lattix LDM to manage a large software architecture. The tool was used to outline specific problem areas in the architecture. The software was then refactored with emphasis on the identified problem areas. The outcome of the refactoring process was validated by comparing the new DSM to the initial DSM.

Creating a DSM of a Java program's dependencies with LDM starts with inputting the data. LDM can analyse Java dependencies from jar, zip, and class files. Once the dependencies have been entered, a DSM is produced. Figure 7 contains the DSM from the packages under org.junit.*. The rows are ordered alphabetically in the initial DSM but the DSM can be partitioned to ease the analysis of the DSM.

LDM's DSMs are nested. This is outlined with the different background colours in the DSM cells of different hierarchy levels. For example, in Figure 7, the user has opened the internal DSMs for three packages under org.junit. The user can analyse the internal DSMs both as separate DSMs and as a part of the whole DSM hierarchy.

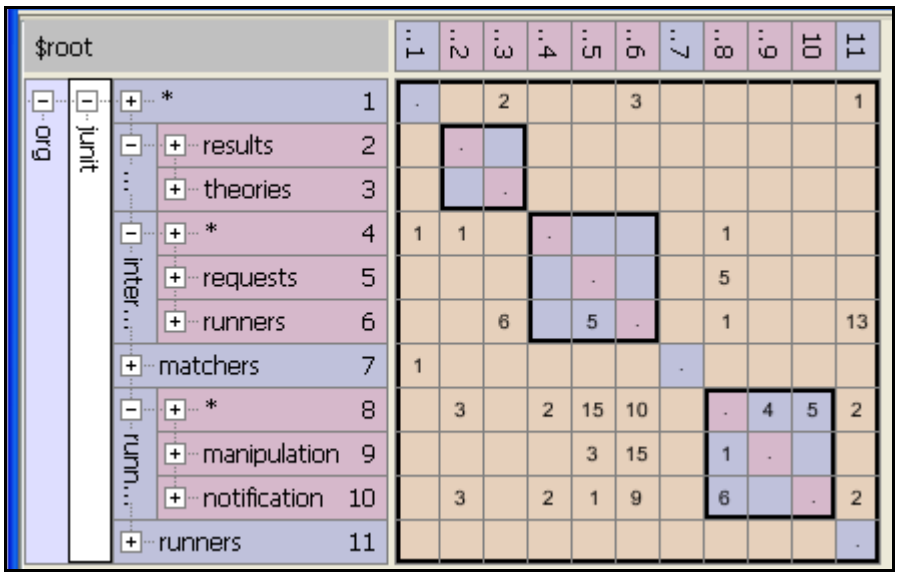


Figure 7. DSM before partitioning.

Figure 8 shows the partitioned version of the DSM. The partitioning resolves whether there are cyclic dependencies in the analysed software. In this case, the six entries above the diagonal tell that there are cycles in the analysed Java packages. The black squares in the partitioned DSM show how LDM groups the different packages into virtual subsystems of interrelated items. The larger the squares are, the greater the need is to refactor the software into smaller modules.

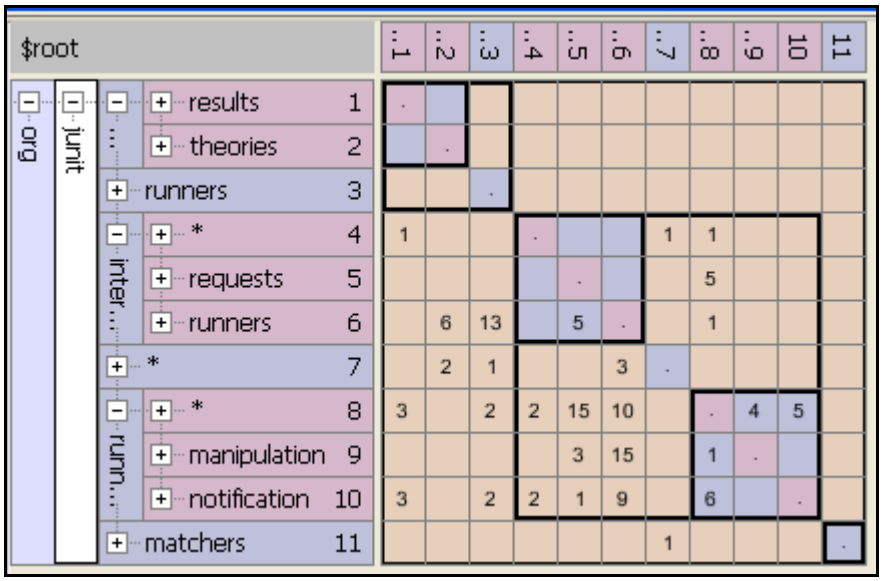


Figure 8. DSM after partitioning.

With Lattix LDM, the software architecture can be refactored directly from the DSM; refactoring the DSM will be reflected in the actual source code of the analysed program. After LDM has partitioned the analysed Java classes into lower triangular form, the areas of the architecture that require the most effort can be identified. The DSM tool will suggest how the software could be refactored to improve its structure. For example, the tool might advise the user to split a Java package with divided responsibilities into two packages, or to move a class from one package to another.

In addition to DSM modelling, LDM shows other details related to dependencies. The user can look at various software metrics generated from the analysed material, including the stability of different packages. The user can also view the type of each dependency in the DSM, for example, whether the dependency originates a constructor call, a method call or due to an inheritance relationship. This is demonstrated in Figure 9. This information screen is always shown next to the DSM, and displays detailed information about whatever DSM cells the user selects. From this example figure, it can be seen that the dependency from org.junit.internal.runners to org.junit is caused specifically by class TestMethod's reference to interface Test.

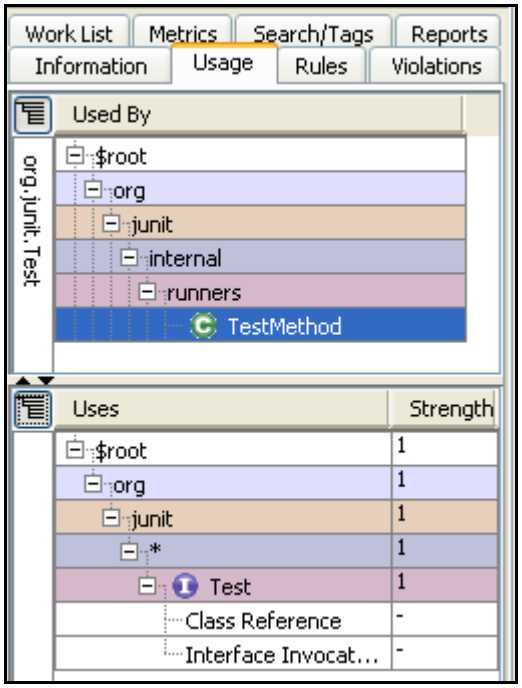


Figure 9. LDM displaying dependency information for a particular DSM entry.

In Figure 10, six cells have dependency rules associated with them. A red triangle on a cell indicates that a dependency rule is violated in that cell. The user can view what rule is broken by clicking on such a cell. Allowed dependencies can be specified as well. LDM allows each dependency by default.

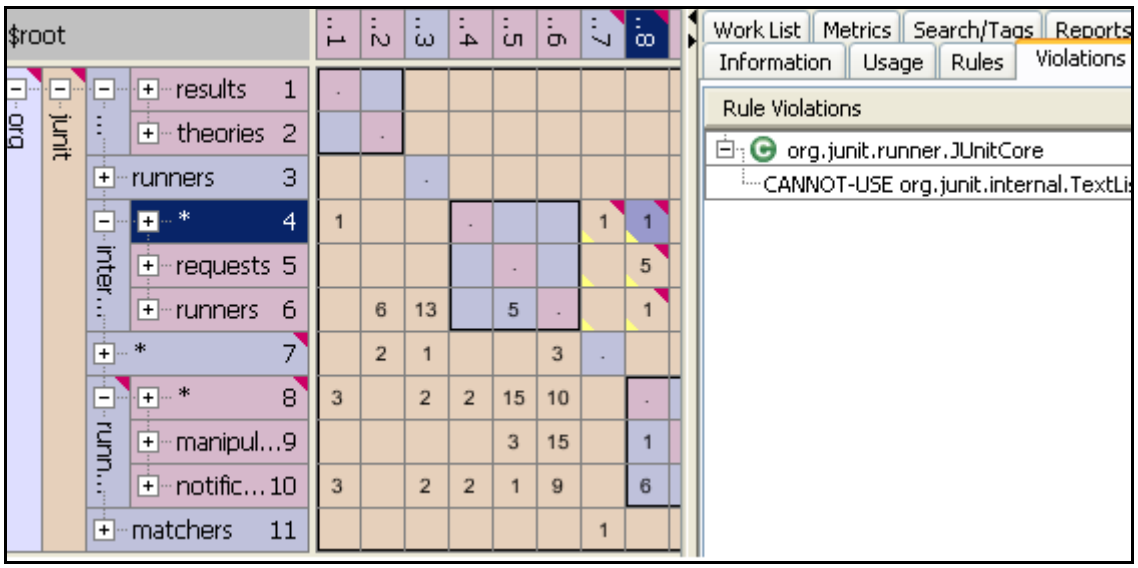


Figure 10. Displaying dependency violations with LDM.

A console version of Lattix LDM, LDC, can be used for integration with automated builds and publishing reports of the analysed material.

4.2.2 Structure101

Structure101 (http://www.structure101.com/) focuses on the structure of software architecture in general, with DSM modelling being one its features. The features related to DSMs consist of partitioning the DSM, highlighting cycles and displaying details about the actual causes of dependencies. Figure 11 shows an example DSM.

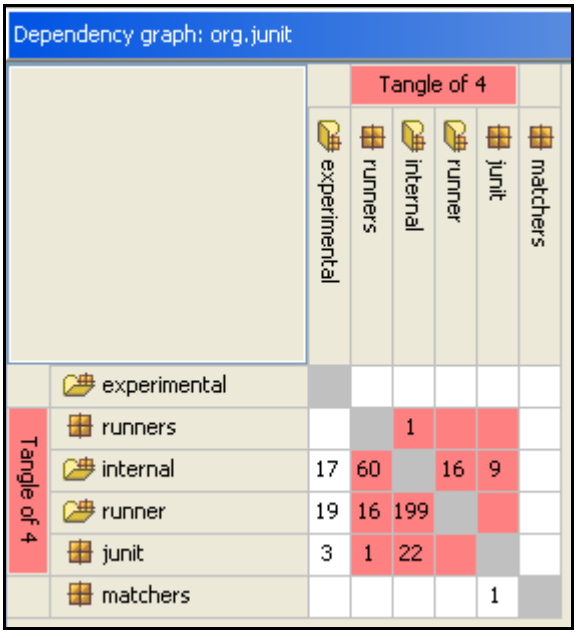


Figure 11. Structure101 displaying a DSM with cycles.

The dependency details can be viewed by selecting the desired DSM cell (see Figure 12). This information includes the names of the dependent and dependee classes, and, when applicable, the method signature of the methods involved. Also, the type of the dependency is specified.

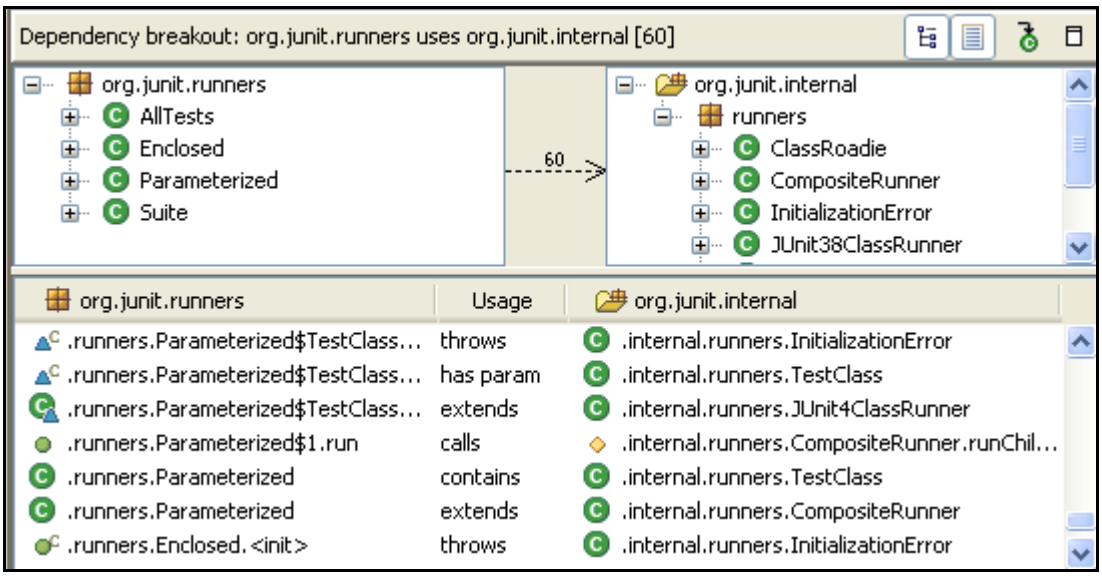


Figure 12. Additional dependency information provided by Structure101.

Even though Structure101 is a commercial product, it is provided free of charge if used with open source software development.

4.2.3 IntelliJ IDEA

IntelliJ IDEA (http://www.jetbrains.com/idea/) is a complete IDE (*Integrated Development Environment*) for Java development and contains DSM generation as one of its features. As can be seen from Figure 13, the DSMs in IntelliJ IDEA differ from the more traditional DSMs. To someone with previous experience with DSMs, reading dependencies can require adjustment because the columns are not named or numbered.

IntelliJ IDEA uses a double crosshair is to show dependencies. This might cause some confusion with the user initially. In this example figure, the dependencies from package ‘ruleanalysis’ to package ‘configuration’ are displayed with the green crosshair and the dependencies in the opposite direction, from ‘configuration’ to ‘ruleanalysis’ are shown with the orange crosshair.

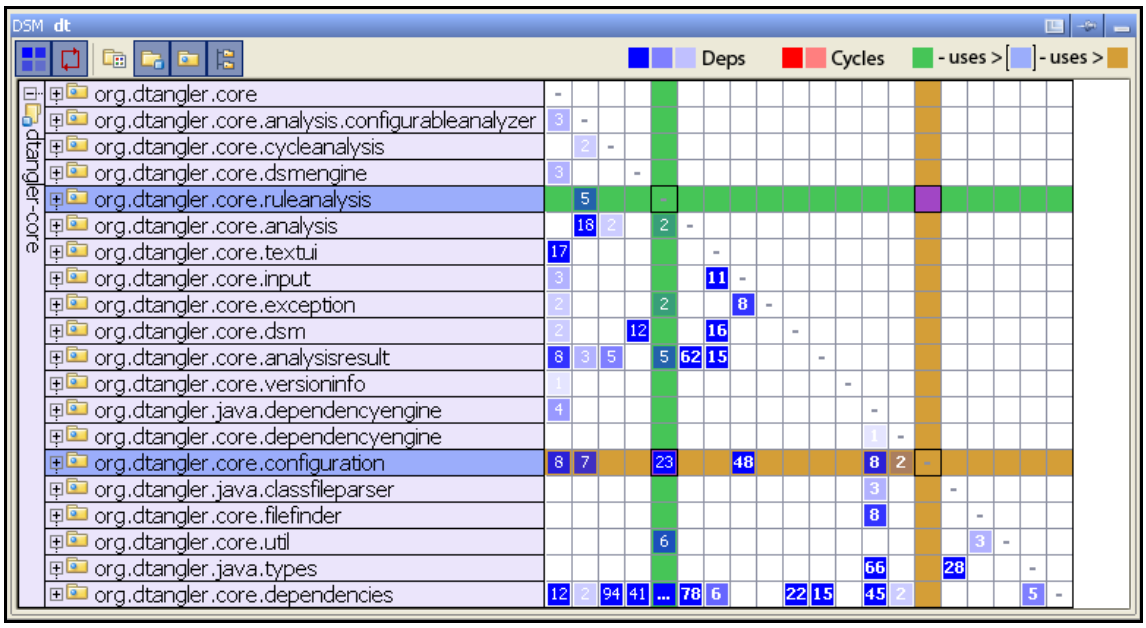


Figure 13. Exploring a dependency structure with the IntelliJ IDEA DSM view.

Even though the DSM display method has its setbacks, the dependency weight modelling is more informative than with other tools. The entries can be highlighted according to the amount of dependencies in a cell. The blue cell background colour darkens when the number of dependencies in that cell increases. Cyclic dependencies are displayed with a red background colour, and the darkness of the colour is also defined by the number of dependencies in the cell.

IntelliJ IDEA's strength lies in it being a complete IDE; this means that the origin of any dependency in a DSM can be pointed directly to the associated source code line. Figure 14 provides an example of this.

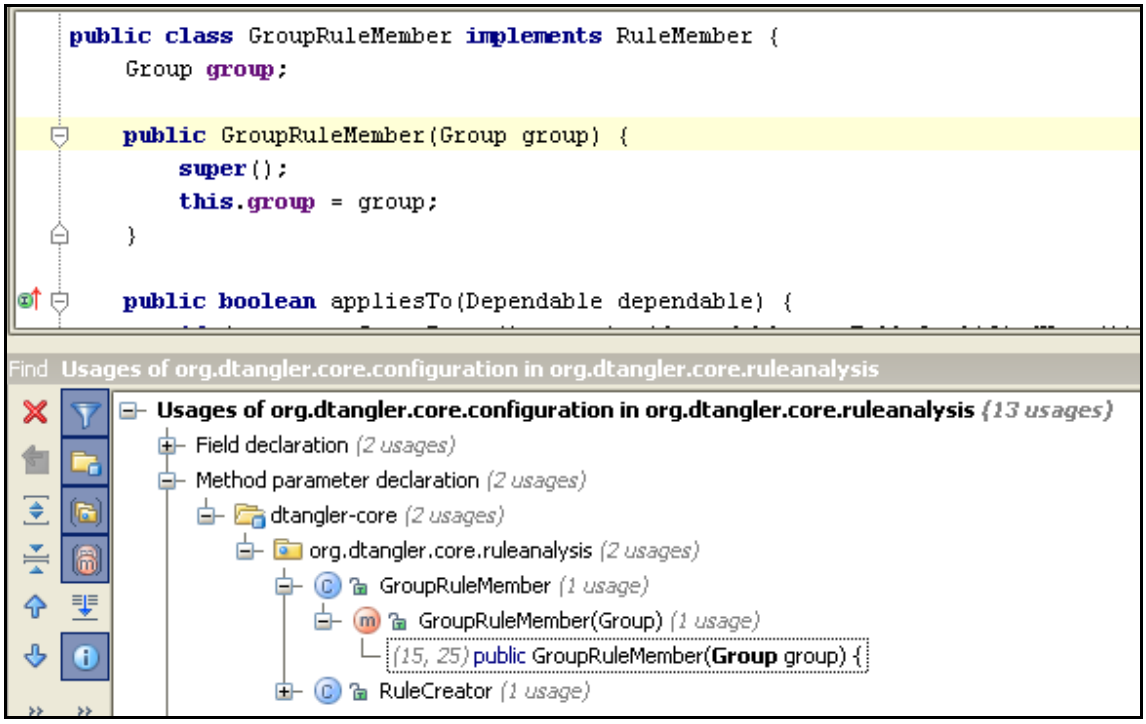


Figure 14. Linking DSM entries to source file lines in IntelliJ IDEA.

One of IntelliJ IDEA's features is to analyse dependencies and specify rules for them. However, because these rules do not affect the DSM display in any way, they are not discussed further.

4.3 Benefits and Problems with the Tools

The use of DSMs in analysing dependencies is very practical in large software projects. The overall health of the dependency structure can be discovered by glancing at its partitioned DSM. All the tools can partition DSMs. LDM is the only one that doesn't do the partitioning automatically. This could pose a problem to someone not familiar with DSMs because the dependency problems are not easily discernible from DSMs without the use of row ordering algorithms.

Detecting cycles is one of the areas in which DSM tools are very efficient. The severity of cyclic dependencies is noted in Structure101 and IntelliJ IDEA by highlighting cycle cell participants with a red colour. LDM doesn't highlight cycles; if there are cycles in the analysed items after partitioning, there are entries above the diagonal. In LDM, cycles could go unnoticed if the user is not familiar with how partitioning works. However, in LDM, cycles are listed in the report generated from the DSM.

Cycles between two items are easy to identify in partitioned DSMs but identifying all cycle participants in long cycles is more problematic. While Structure101 and IntelliJ IDEA highlight cells participating to cycles with a red background colour, this does not tell the user how many cycles there are and what the exact cycle paths are. The cycle report generated by LDM doesn't reveal specific cycle paths, either.

Structure101 and IntelliJ IDEA do not integrate dependency rules in their DSMs. LDM provides the user a lot of additional value through the use of dependency rules. It makes it possible to communicate the high-level architecture between developers. Any breaches to the defined rules are identified as soon as they are made and corrective actions can be taken right away. The end result might be a correction to the implementation that broke the rule, or a modification of the dependency rule itself. Either way, the implementation remains consistent with an articulated design.

However, rules in LDM have one setback: you can only attach rules to items already present in the analysed material. This means that dependency rules cannot be entered and monitored until there is an actual implementation from which to extract a DSM. This inhibits the tool's usage proactively as dependency rules cannot be applied to architectural decisions right from the start of the implementation.

4.4 Summary

While there are numerous non-commercial tools available for analysing dependencies, they don't support modelling dependencies in DSM form. Dependencies are usually described textually, or with different variations of box-and-arrow graphs. There are only a few DSM tools developed in research, and the information available about these tools is scarce. However, some commercial DSM tools exist and can be examined by downloading trial versions. The main features of three tools are summarised in Table 5. The core functions of all the DSM tools are to analyse dependencies from defined input sources and to deliver an ordered DSM representation of these dependencies. LDM also supports dependency management with its capability to define rules for the dependencies.

Table 5. Comparison of features between commercial DSM tools.

Feature	Lattix LDM	Structure101	IntelliJ IDEA
Dependency rules	Yes	No	No
Dependency information	Type of dependency is shown	Detailed dependency information is shown	Code causing the dependency is shown
Dependency input sources	Ada, C/C++, Delphi Pascal, Java, .NET, Oracle, SQL Server, Sybase, XML for generic dependencies	Java, C/C++, XML for generic dependencies	Java input from the IDE's projects
Refactoring software from the DSM	Yes	No	No
Hierarchical DSMs	Yes, in nested DSMs	Yes, in separate DSMs	Yes, in nested DSMs

The main difference between the three DSM tools presented is in their focus. LDM concentrates solely on modelling dependencies with DSMs, IntelliJ IDEA is a complete IDE that includes DSMs as a part of it, and Structure101 focuses on software structure but not specifically on DSMs. While you can display dependencies in DSM form with all three tools, Lattix LDM offers the widest set of features for dependency management with DSMs.

While the tools make it easier to analyse software dependency structures, there are still some issues to be solved. The integration of dependency rules is a good step towards integrating dependency management into the active development phase. This could be further improved by integrating and comparing design-time DSMs with implementation level DSMs. Cai and Hyunh (2006; 2007) have addressed this matter more closely in their research.

5. Implementing a Dependency Structure Matrix Tool: Dtangler

This chapter presents dtangler, an open source software product for dependency analysis and management with DSMs. The process methods of this project are described, and the goals of the project are listed. The main focus, however, is on implementation aspects. Finally, the experiences of the project and future directions are discussed.

5.1 Background

The development of dtangler started in the beginning of 2008, and is based on a previous prototype version. The preliminary version featured a command line user interface, and produced a DSM by analysing Java package level dependencies. This chapter discusses dtangler version 1.2.0.

In an ideal situation, dtangler is integrated with the whole software development process. Software architects can specify dependency rules for the software being developed. The compliance to these rules can then be monitored, preferably by integrating dtangler with automated builds.

The tool is open source, and can be downloaded from http://www.dtangler.org/. The web site also contains material related to the tool's development, including the Scrum product backlog and detailed sprint information.

5.2 Description of the Implementation Process

The development of dtangler uses Scrum and Test-Driven Development (TDD). The main characteristics of these methods are summarized below.

Scrum is an agile software development process that creates functional software versions in short iterations, called sprints. A sprint usually lasts a couple of weeks. Scrum roles consist of product owner, scrum master, and team. The product owner is responsible for defining and prioritising requirements, and accepting or rejecting sprint results. The scrum master's role is to facilitate the actual development done by the team members, for example by serving as a mediator between the team and anyone else. The team is responsible for the actual work of completing the tasks set for each sprint. (Koch, 2004, Appendix H; Schwaber, 1995.)

The main idea behind TDD is that writing tests for a feature is done before implementing the actual feature. Development with TDD is a three-step activity. First, an automated test for the required feature is written. Next, the feature is implemented so that the test is passed. The last step is to refactor both the test and the implementation. Writing the test first helps the developer to define more precisely what the new feature should do. Also, having a large base of automated tests helps verify that making changes to the code hasn't broken the existing functionality. (Janzen & Saiedian, 2005.)

Up until version 1.2.0, the dtangler project line-up consisted of a product owner, a scrum master, and a small team, whose size varied from one to three team members.

The author of this thesis was a team member starting from sprint 1, and implemented features to the dtangler core. There were 3 major releases and eight sprints in approximately six months of development.

In addition to integrating unit tests and implementation with TDD, writing user documentation is integrated to writing acceptance tests. The idea is similar to TDD but in this case, you first write the user instructions for a particular feature, and then write the acceptance test that proves that this feature does indeed work as described. The Bumblebee tool (http://www.agical.com/bumblebee/) is used to generate the user documentation from the test files. Bumblebee also enables the use of run-time data as a part of the documentation. By keeping the tests and user documentation close together like this, the user documentation always reflects the current state of the software instead of being possibly outdated.

5.3 Objectives

The requirements set for dtangler are summarized in Table 6, and described in more detail below.

Table 6. Requirements for dtangler.

Features	Dependency extraction from Java class and jar files
	Analysing dependencies on multiple detail levels
	Presenting dependencies in DSM form
	Dependency rules
Implementation details	Existing dependency metrics are utilised (coupling, cohesion, stability)
	DSM rows are ordered with a selected algorithm
	The dependency structure of the tool itself is in good shape
Other	The tool is open source

The aim of dtangler is to help software developers' work by providing a way to manage and visualise software dependencies. One of dtangler's goals is to be a considerable open source alternative for similar commercial tools. To achieve this, dtangler should be able to perform the same core functionality as commercial tools. These features were studied more closely in Chapters 4.2 and 4.4, and include the extraction, analysis, presentation, and management of dependencies.

Dependency analysis should take advantage of existing dependency metrics. As described in Chapter 3, coupling is an important metric to consider when analysing dependencies between modules on the same detail level. Cohesion should be considered when analysing dependencies on different detail levels. This basically means that the internal dependencies of, for example, a single Java package have to be analysed and presented. Instability measuring should be used to further analyse the quality of the dependencies.

When modelling dependencies in a DSM form, the main problem is how to order the rows in the matrix. In previous research (see Chapter 2.4), either clustering or partitioning algorithms have been used to sort the rows on the matrix. Partitioning

outlines software layering, while clustering is mostly useful in analysing the optimal workflow of product development by grouping interrelated items on the DSM together (Yassine, 2004, p. 13-14).

Another point worth mentioning is that when developing a tool to manage dependencies, it is imperative for the tool itself to have a healthy dependency structure. This adds to the credibility of the product, and is even further important because the code is open source and thus available for inspection by anyone.

5.4 Features

The main features of dtangler include the following:

- Dependency analysis from Java classes and jar files
- DSM output on multiple detail levels (file location, package, and class levels)
- Dependency rule configuration for single items and groups of items
- Cycle and rule violation detection
- Saving and loading configurations

When starting dtangler, the user can either select the input folders for the analysis manually, or load a previously created properties file. The input tells dtangler where to search for the material to be analysed. The supported file formats are Java class and jar files. The selected file locations make up the topmost scope for the DSM. This can be useful when analysing dependencies between different projects, for example.

The tool comes with two user interfaces, a graphical user interface, and a console user interface. The GUI (G*raphical User Interface*) is aimed for examining dependencies of existing software. This can be useful when refactoring the software, for example. The GUI can also be used to edit dependency rules and other properties.

The GUI main view is presented in Figure 15. In this example, the user has input three file locations, and the dependencies between these locations are shown. The user can switch between full and shortened DSM row names by clicking on the green ‘n’ button. The number in parentheses after each row name tells how many items that row item contains. In this case, the core file location contains 21 packages.

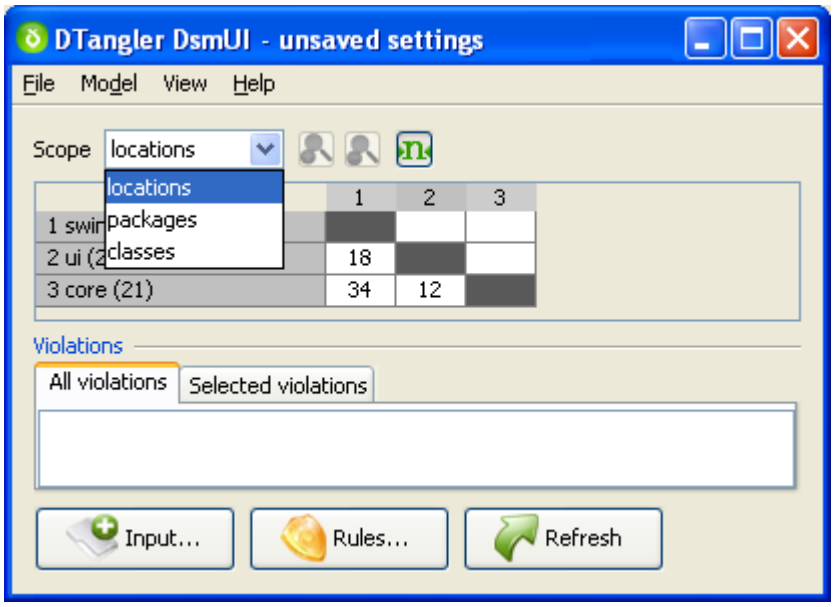


Figure 15. The graphical user interface of dtangler.

A console version of the same input material is shown in Figure 16. The text-based user interface is aimed for integrating dtangler with automated builds. Its return value tells whether or not the analysed material contains any dependency violations.

```
dtangler 1.2.0 (c) 2008 by contributors.
check www.dtangler.org for new versions and additional information
                                                    |   1|   2|   3|
  1 ..tangler-swingui\_build\eclipse\classes ( 26) |####|    |    |
  2 ..src\dtangler-ui\_build\eclipse\classes (  2) |   18|####|    |
  3 ..c\dtangler-core\_build\eclipse\classes ( 21) |   34|   12|####|
```

Figure 16. The text-based user interface of dtangler.

Cycle detection is demonstrated in Figure 17. Cells with a red background participate to cycles on the current scope. The red background on a row name indicates that there is a cycle inside that item, in this case, within the classes inside the outlined package. The problem inherent to DSMs is that it is difficult to analyse long dependency paths directly from the matrix. To ease the analysis of cyclic dependencies in dtangler, the cycle paths are listed in the Violations information screen below the DSM. The user can also examine cyclic dependencies related to any single DSM cell by selecting the desired cell and the ‘Selected violations’ tab. This will filter out any irrelevant cycle path listings from the violations screen.

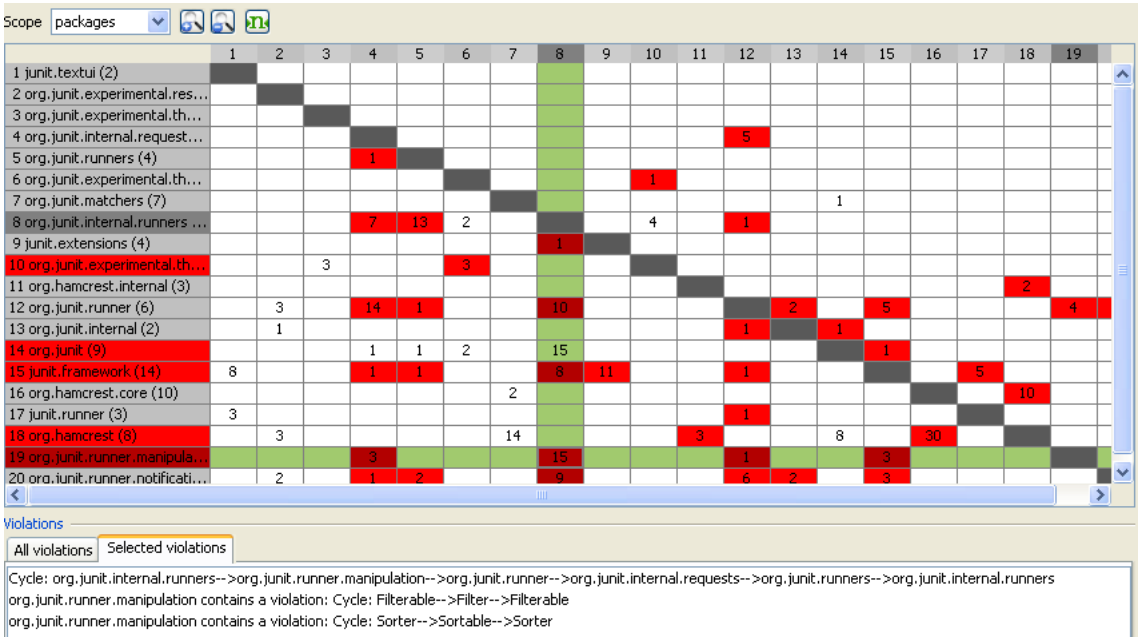


Figure 17. Cycles are displayed with a red background colour.

The user can add dependency rules to the DSM. In the DSM of the previous figure, the user might want to forbid the dependency from item 14 to item 7 because this cell is quite far above the diagonal. After adding a rule that denies this dependency, the cell background turns yellow (see Figure 18). The violation list also shows what specific rule is broken.

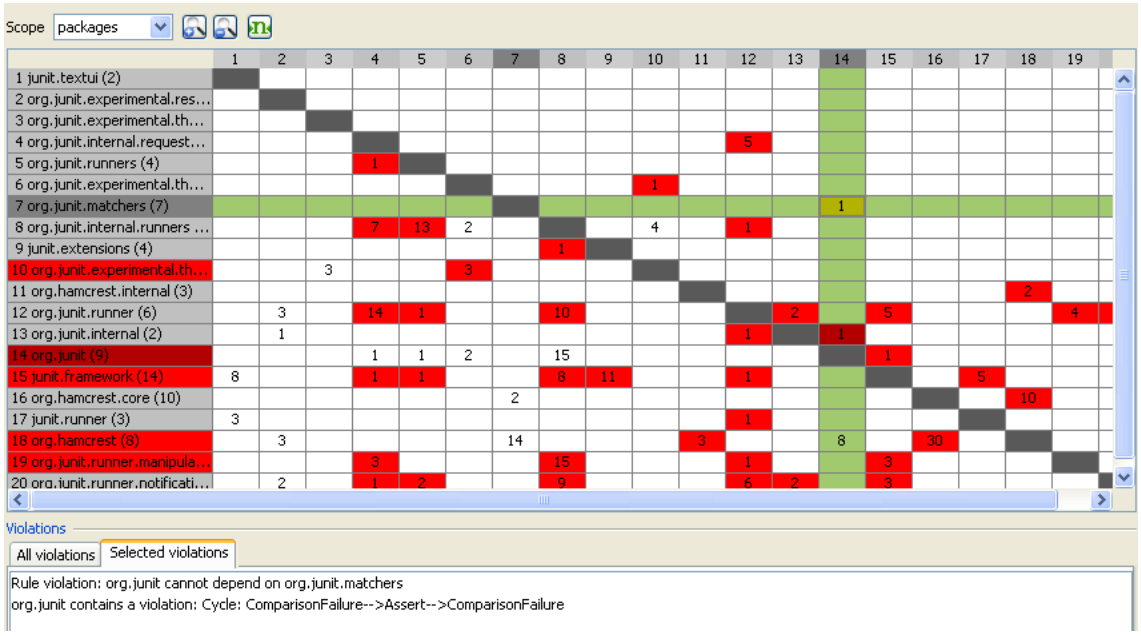


Figure 18. Forbidden dependencies are displayed with a yellow background colour.

It is also possible to set dependency rules for groups of items, and to use rules to allow dependencies. Forbidden dependencies are always overridden by matching allowed dependencies. For example, the user might not want test class dependencies to show up as violations. If the tests are under the package foo.bar.tests and its sub-packages, the user could create a ‘Tests’ group that contains items whose name match ‘*foo.bar.tests*’, and an ‘All items’ group that contains all items, ‘*’. The user could

then specify that Tests can depend on All items, and thus make the DSM show only those violations that are relevant to the user.

The settings for each project can be saved to a properties file. The properties files are useable by both the GUI and the console version of dtangler, and can also be edited manually.

5.5 Design and Implementation

This section presents the implementation decisions most relevant to this research. After the high-level architecture has been outlined, the extraction, handling, and displaying of dependencies is described.

The focus of this section is specifically on the design and implementation of dtangler's core module. This is due to the facts that the core module contains the essential functions for dependency management, and because the author of this thesis was not involved in the design and implementation of the user interface modules.

5.5.1 High-Level Architecture

The architecture of dtangler consists of three main modules: a generic user interface module, a Java Swing user interface module, and a core functions module.

The high-level architecture is presented in DSM form in Figure 19. The figure shows that the Swing UI module depends on the both the UI module and the core module, while the UI module depends only on the core module. The core module depends on neither the UI nor the Swing UI module. Layering the software in this manner makes it possible to replace the specific Swing UI module with a UI module of another type without causing any changes to the other two modules, UI and core.

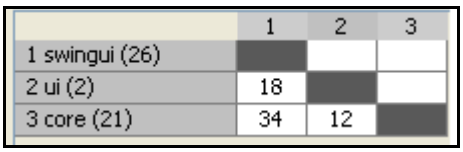


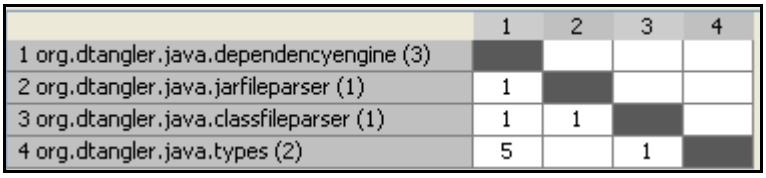
Figure 19. High-level architecture of dtangler, produced by dtangler.

The core module can be further divided into two sub-modules: the actual core that handles the dependency management of Dependable objects, and a Java module that encapsulates anything related to creating dependencies from Java input. The idea behind this separation is to make the core able to model dependencies from any input source. For example, a module that extracts dependencies from C++ files could be added, and handling these dependencies in the core would be no different from handling dependencies originating from Java files.

5.5.2 Dependency Extraction

The Java module is responsible for parsing dependencies from Java class files. Java class files are binary files, generated by a Java compiler from Java source files (Lindholm & Yellin, 1999, Chapter 4).

Figure 20 describes the dependency structure of the various packages involved in extracting Java dependencies. The dependency engine package provides the interface for setting the input path from which class and jar files are searched. The engine finds the files and passes the data on to either the jar file parser or to the class file parser.



	1	2	3	4
1 org.dtangler.java.dependencyengine (3)				
2 org.dtangler.java.jarfileparser (1)	1			
3 org.dtangler.java.classfileparser (1)	1	1		
4 org.dtangler.java.types (2)	5		1	

Figure 20. DSM produced from dtangler's Java packages.

If jar files were found, their contents are read to find class files inside the jar archive. These class files are then passed on to the class file parser as byte streams. Class files not inside a jar archive can be passed to the class file parser directly. The class file parser searches binary class files for dependency information from the class' methods, fields, constant values, super class and interface definition.

The end result of parsing is a set of JavaClass instances. JavaClasses, which reside under the types package, are used to model the relevant aspects of parsed Java binary classes. A JavaClass instance contains information of the dependencies and dependency weights of this particular class, along with the name of the class, the package, and the location of the file. The dependency weight simply represents the number of references this class has to the dependee class.

When the JavaClass instances have been created, the dependency engine transforms them into generic Dependable objects and maps the dependencies between them. The dependencies of class Dependables' parents, packages, jars, and file locations, are also resolved. This process is explained in more detail in the following section.

5.5.3 Handling Dependencies

The challenge of handling dependencies arises from the requirement of having dependencies on multiple detail levels, scopes. Basically, this means that Dependable objects need to be grouped by their scope, and that each Dependable can be mapped to either a parent Dependable, any child Dependables, or both.

Figure 21 shows an example of how scopes should work. The user has input two folders to the analysis, C:\ProjectA and C:\ProjectB. The dependencies between the classes in these two locations are shown in the topmost DSM. Project B is the parent Dependable for the three packages displayed in the middle DSM. Likewise, b.package3 is a parent to the three classes in the bottom DSM. To analyse dependencies correctly, the program has to be able to navigate from one Dependable to the related Dependables on other scopes.

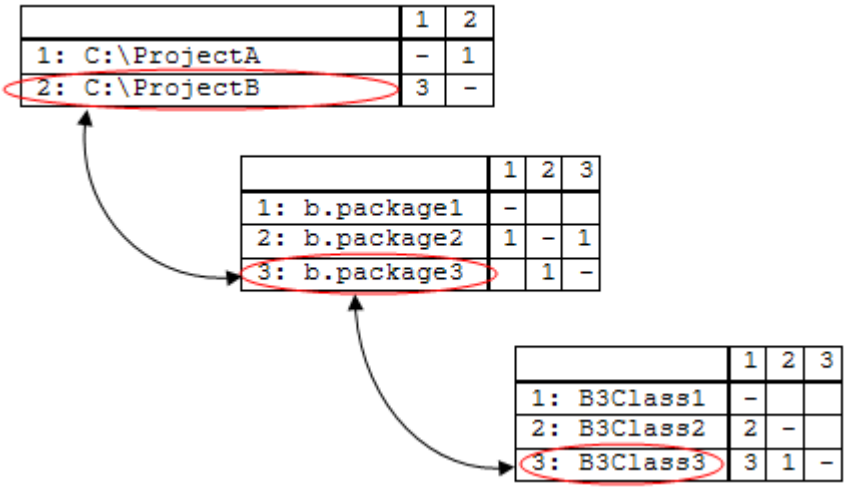


Figure 21. DSM relations between scopes.

All dependencies are resolved at the lowest scope, in this case, the Java class level. Package level dependencies are calculated by summing up the dependencies of each class contained in a specific package. In turn, file location level dependencies are resolved by analysing the dependencies of each package contained in that specific location.

The functions related to dependency and detail level handling are contained in package org.dtangler.core.dependencies. The class level DSM of this package is presented in Figure 22.

	1	2	3	4	5	6	7
1 Dependencies (2)							
2 DependencyPath (1)							
3 DependencyGraph (1)	8						
4 ScopeComparator (1)	1						
5 Dependency (1)		3	2				
6 Dependable (1)	19	5	16		7		
7 Scope (1)	16		4	3		3	

Figure 22. DSM for the classes inside the dependencies package.

Class Dependencies handles dependencies on all detail levels, scopes. It maps Dependables to their parents and children. To manage this, Dependencies keeps a list of all possible scopes. At the moment, all scopes in dtangler are Java-related: packages, classes, and class or jar file locations. The row name Scope in the DSM of Figure 22 refers to an interface called Scope; it defines that a scope should be able to return its name and index. The class JavaScope, from the org.dtangler.java.types package, is just one implementation of this interface. The separation of the general interface Scope and its implementation, JavaScope, is one of the ways how general dependency issues are kept apart from Java-specific dependency issues in dtangler.

Apart from handling scopes, Dependencies contains a listing of dependencies and dependency weights between all the analysed items. DependencyGraph contains dependencies on a single scope. Any material on a DSM produced by dtangler comes from a DependencyGraph. When looking at Figure 21, Dependencies is related to the image as a whole, DependencyGraph to a single DSM, and Dependable to a single item on a DSM.

5.5.4 Generating DSMs

Modelling dependencies in a matrix form is quite straightforward: a DSM class contains instances of DSM rows, which in turn contain DSM cells. This section explains the DSM row ordering in detail.

The algorithm used to order the DSM rows is a modification of the partitioning algorithm. In the traditional partitioning algorithm, an entry above the DSM diagonal is always an indication of a dependency cycle. In dtangler, cycles are highlighted by marking the cells participating in a cycle.

Displaying cyclic dependencies in this manner has the advantage of showing more information to the user. This method shows all items participating to the cycle; partitioning only shows the existence of a cycle.

In other words, dtangler a DSM may contain entries above the diagonal even when there are no cycles between the analysed items. With dtangler, this is rather an indication that the items are not properly layered.

The rows are ordered in the following manner. The main ordering criterion is the instability of the items. The instability is calculated according to Martin's (1997) formula. Figure 23 demonstrates this concept. Items with low stability are placed topmost on the DSM. In short, depending on a more stable item is acceptable. All entries on the lower triangle are considered healthy dependencies, indicated with a green background colour in the image. Entries on the upper triangle are unhealthy, and increasingly so the further they are from the diagonal. This principle and Martin's instability metric were discussed in more detail in Chapter 3.2.

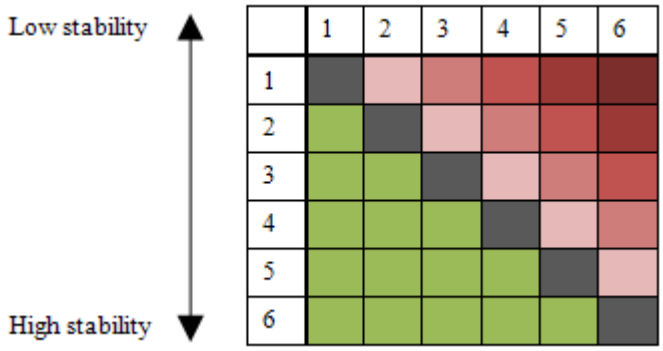


Figure 23. DSM rows in dtangler are ordered by their stability.

Further row ordering is required if there are items with equal stability values. As recalled, instability is calculated with regard to how many other items the item under analysis depends on, and how many other items depend on that item.

The instability calculation doesn't take into account the dependency weights of any particular dependency. If there are equally instable items, the dependency weights of the items' outgoing dependencies are summed up. The item with the largest amount of outgoing dependencies is considered to be the least stable. For example, consider a case where A has ten references to C, and B has only one reference to C. If C were to change, A is most likely required to go through more changes than B.

In the case of equal instability values and equal outgoing dependency weight totals, the rows are ordered alphabetically.

5.6 Experiences and Future Directions

Scrum was a good choice for a project like this because the idea behind dtangler wasn't set in stone when the project started. Many new requirements and improvements were discovered as the project progressed. With Scrum, the product owner re-evaluates and prioritises requirements constantly. The features implemented in the previous sprint brought up new ideas about how the software should work. As a developer, working in short sprints to complete atomic, well-defined tasks was more motivating than just working toward one large goal somewhere in the future.

One of the requirements addressed the need for a good dependency structure in dtangler. This made Java a good choice for the implementation as it allowed the evaluation of dtangler's dependencies with dtangler itself. Other than that, the tool probably could have been developed in a similar fashion with any object-oriented language.

The hardest part in the implementation was probably modelling dependencies and handling dependency rules on multiple scopes. When searching for dependency violations, the violations on lower scopes have to be somehow visible on the upper scopes as well. Also, allowed dependencies and grouped items added more difficulty to the rule analysis.

Many interesting features were left out of dtangler due to limited resources. Also, the implementation was directed towards features most beneficial to addressing the research questions set in this thesis. Fortunately, the development of dtangler doesn't end with this research. There are at least two larger areas that could be developed in the future: analysing dependencies of items outside the Java domain, and creating a plug-in for Eclipse.

5.7 Summary

The development of dtangler was done with agile software development methods, using the Java programming language. The main goal of the tool was to provide a good open source alternative to commercial DSM dependency analysis tools.

The tool evolved over a period of six months from a prototype DSM dependency modelling tool to a complete dependency management tool. The most significant additions were dependency rules and multi-level dependency analysis. These features support the idea of making sure that the dependencies in a project are in good order throughout the software's life cycle.

The DSM ordering algorithm used in dtangler should be highlighted. It is a modification of the traditional partitioning algorithm. While both algorithms ideally produce a lower triangular DSM, the approach used with dtangler is a bit stricter. With partitioning, the DSM will always be lower triangular if there are no cycles between the analysed items. In dtangler, in addition to no cycles, no item in the DSM can depend on a less stable item than itself.

Using the modified partitioning algorithm to sort the DSM rows provides more information to the user about the software under analysis. Even though cyclic dependencies are agreed to be the worst kind of dependencies, a software program with no cycles is not necessarily a model example of good design.

6. Evaluation of Dtangler

The constructed dependency analysis tool is evaluated in this chapter. The evaluation is divided into three parts: evaluation against the tool's own requirements, comparing dtangler to a similar tool, Lattix LDM, and analysing end users' experiences of dtangler.

LDM was chosen as a point of comparison out of the tools reviewed earlier in this research because it resembles dtangler the most: both tools focus on DSM modelling and have built their features around DSMs. In addition to having the most extensive set of features, the selection of LDM was favoured due to its academic evaluation license.

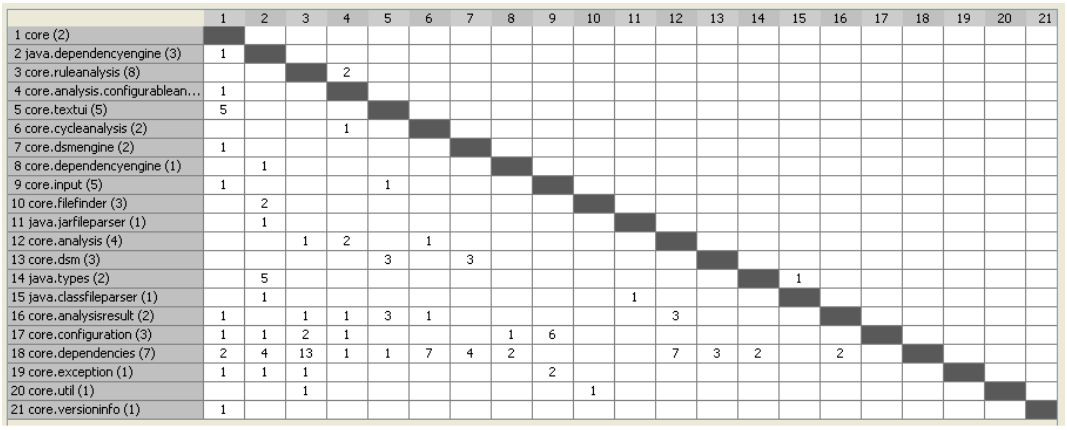
6.1 Conformance to Requirements

The requirements for dtangler were presented in the previous chapter, in Table 6. In addition to feature requirements, the tool was required to use a DSM ordering algorithm, use existing dependency metrics, have a healthy dependency structure, and to be open sourced. The following section discusses how these requirements are met.

All of the required features were implemented. They are discussed further in Chapter 6.2 when comparing them to LDM's features.

DSM row ordering is done with a modification of the partitioning algorithm. The modified algorithm is based on Martin's (1997) stability metric. Coupling of items is presented in the DSM cells that connect these items. The cohesion of a module can be analysed by looking at the internal DSM of the module: the DSM of a highly cohesive module is lower triangular. However, the tool does not analyse the cohesion of items on the lowest scope, the Java classes. Because DSMs are more aimed towards dependency modelling on higher detail levels than single classes, a different approach would be needed to measure class level cohesion.

DSMs produced from dtangler, by dtangler, were evaluated throughout the development process to identify possible unwanted dependencies, especially cyclic dependencies. Thought was also placed on separating the user interface and the core functionality from each other. The DSM for the core module is presented in Figure 24. The DSM is not completely clean because there are two entries above the diagonal. Even though this indicates that the dependency structure is not perfect, the DSM is not extremely alarming because both upper triangle entries are right next to the diagonal. The situation would be different if there were cyclic dependencies or a lot of entries above the diagonal.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1 core (2)																					
2 java.dependencyengine (3)	1																				
3 core.ruleanalysis (8)				2																	
4 core.analysis.configurablean…	1																				
5 core.textui (5)	5																				
6 core.cycleanalysis (2)				1																	
7 core.dsmengine (2)	1																				
8 core.dependencyengine (1)		1																			
9 core.input (5)	1				1																
10 core.filefinder (3)		2																			
11 java.jarfileparser (1)		1																			
12 core.analysis (4)			1	2		1															
13 core.dsm (3)					3		3														
14 java.types (2)		5													1						
15 java.classfileparser (1)		1									1										
16 core.analysisresult (2)	1		1	1	3	1						3									
17 core.configuration (3)	1	1	2	1				1	6												
18 core.dependencies (7)	2	4	13	1	1	7	4	2				7	3	2		2					
19 core.exception (1)	1	1	1						2												
20 core.util (1)			1							1											
21 core.versioninfo (1)	1																				

Figure 24. DSM of dtangler's core

The tool is open source, and is provided under the terms of Eclipse Public License (The Eclipse Foundation, 2004).

6.2 Feature Comparison

Table 7 presents a comparison of the features in dtangler and LDM for Java. Differences in the offered features are analysed more closely below.

Table 7. Features of dtangler and LDM compared.

Category	Feature	dtangler	LDM
Dependency input	.class	Yes	Yes
	.jar	Yes	Yes
	.zip	No	Yes
Dependency management	Rules for groups and single items	Yes	Yes
	Rule definition prior to implementation	Yes	No
	Direct refactoring	No	Yes
	Saving and loading configurations	Yes	Yes
Dependency analysis	Cycle detection	Yes	Yes
	Rule violation detection	Yes	Yes
	Dependency details	No	Yes
DSMs	Row ordering	Yes	Yes
	Nested DSMs	No	Yes
	DSM filtering	No	Yes
Dependency detail levels	Input file locations	Yes	No
	Packages	Yes	Yes
	Classes	Yes	Yes
	Class members	No	Yes

From the comparison it is visible that LDM has a wider set of features than dtangler. The ZIP file format is not supported with dtangler's dependency input. One of the biggest differences is that unlike dtangler, LDM offers direct refactoring. This means that the architecture can be modified by manipulating the items and their hierarchy on the DSM. Another issue is that LDM can display the type of each dependency while dtangler only displays the names of the dependent and the dependee.

The dependency analysis in LDM goes down to the level of class members; dtangler's lowest DSM scope is class level. On the other hand, the usefulness of analysing dependencies inside single classes with DSMs is questionable. In object-oriented programming, keeping the classes reasonably small is considered a good programming practice (Fowler, 1999, p. 78). DSMs are mostly suited for situations in which there are a lot of items to display.

Even though LDM doesn't support input file locations as individual DSM items in the sense that dtangler does, the user can group the DSM items into different parent sub-systems with LDM. The end result is similar in both cases.

The DSM related operations in dtangler are limited to row ordering. In LDM, the DSMs are nested (see Figure 25), and items can be hidden from the DSM completely. With dtangler, the DSMs of different detail levels are always shown on separate DSMs, without nesting.

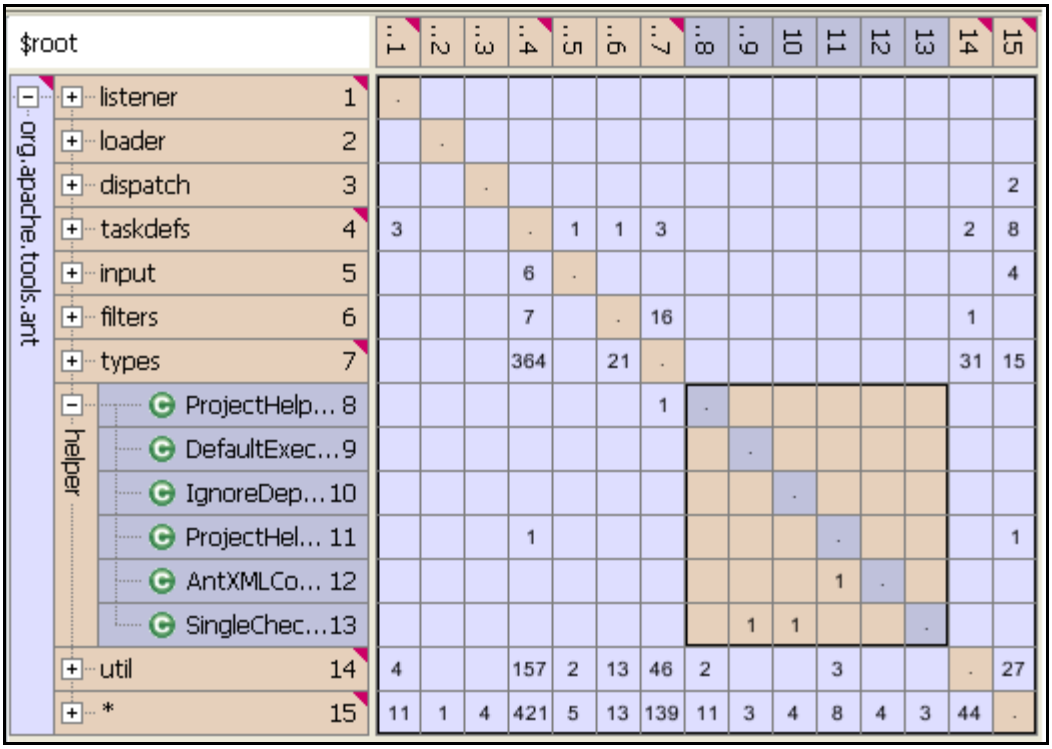


Figure 25. LDM nests DSMs on different detail levels to the same DSM.

Feature-wise, one benefit dtangler has over LDM is that dependency rules can be specified before any sort implementation is in place. This can be useful in laying out rules for the high level architecture. For example, when the project's sub-modules or a general package structure has been designed, their names can be used to specify groups. These groups can then be used to define how modules or packages can or cannot depend on each other.

6.3 Analysis Result Comparison

The differences in how dtangler and LDM function in practice are analysed in section. The analysis is performed by inputting the same material to both programs and by comparing their output. JUnit 4.4's (http://www.junit.org/) core packages, org.junit.*, serve as the input material. JUnit was chosen because it is open source and relatively small. This makes it easier to analyse the causes of possible differences in the produced DSMs.

6.3.1 DSM Output

Figure 26 shows dtangler's output for classes under org.junit. All violations are listed below the DSM and highlighted with a red colour in the DSM cells and rows. In this case, all the entries above the diagonal are cycle participants but as recalled from the previous chapter, there could also be entries above the diagonal even if they were not included in any cycle. These entries would only indicate a dependency in the wrong direction: a stable item depending on an instable item.

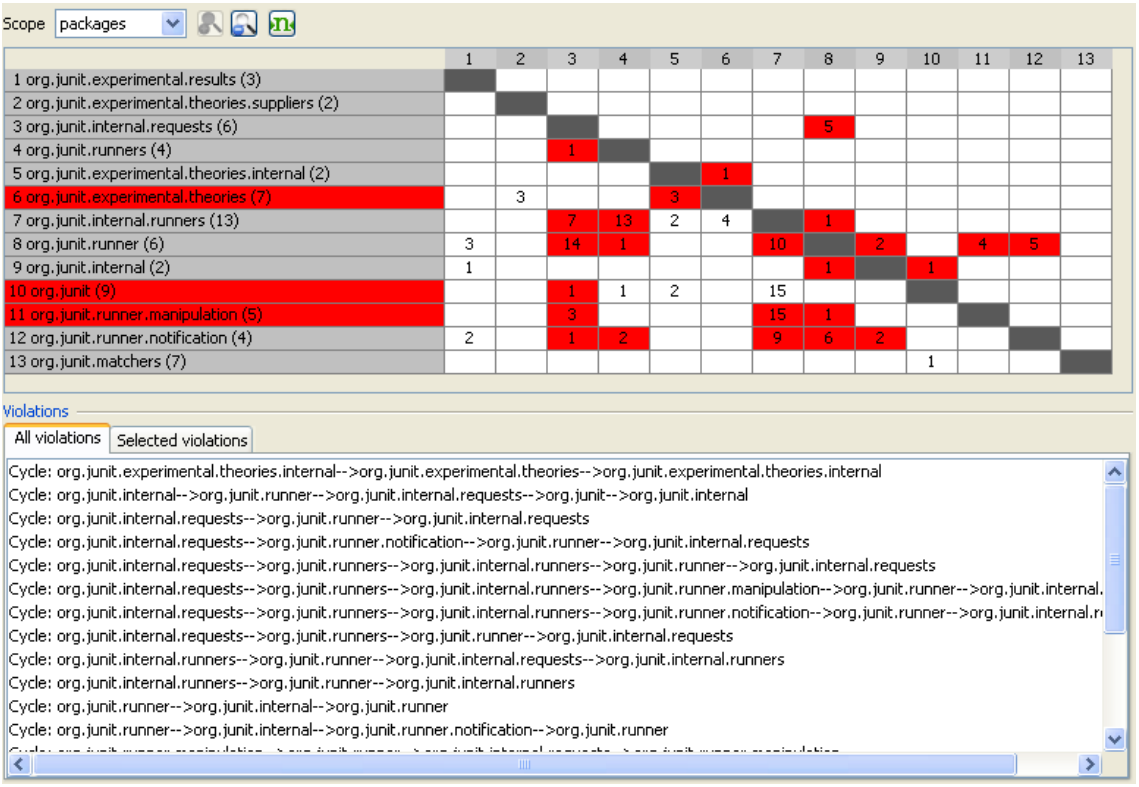


Figure 26. Displaying JUnit's dependencies with dtangler.

The DSM produced by LDM is presented in Figure 27. This DSM is not what the software initially outputs; it has been partitioned first. In LDM, black squares are used in partitioned DSMs to visually group interrelated items. They suggest what items each software module should contain to avoid cyclic dependencies between modules. In effect, this encapsulates all cycles inside single modules. If the system is not well structured, the size of the suggested modules can grow up to the point that the whole DSM is just one big square.

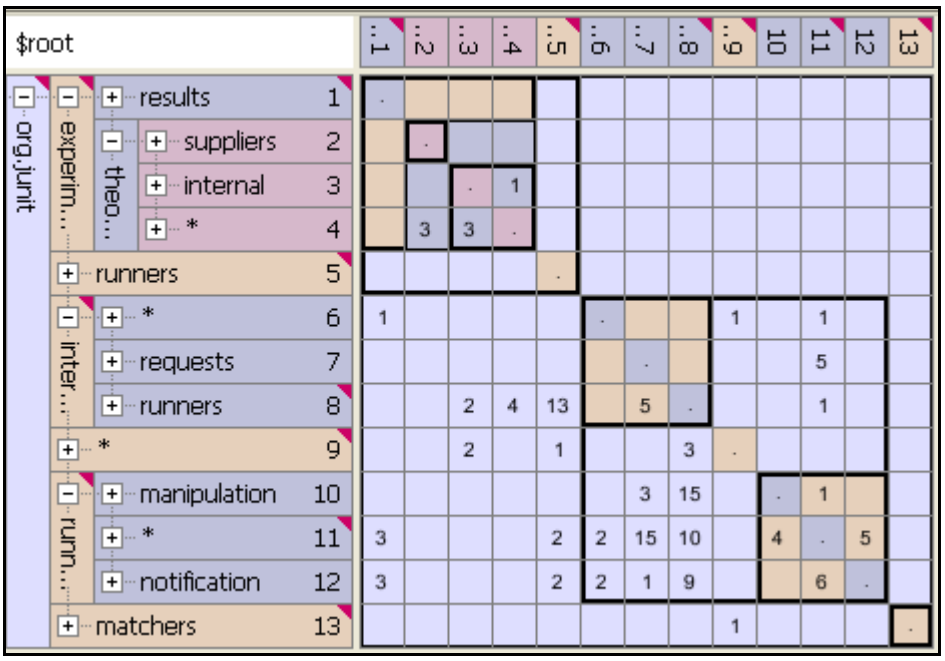


Figure 27. Displaying JUnit's dependencies with LDM.

The main difference between the tools' DSM output is that LDM nests DSMs of different levels to the same DSM and dtangler displays separate DSMs for each detail level. LDM's version is easier to navigate because the user always sees the complete hierarchy on the left of the DSM. Another benefit is that multiple lower scope DSMs can be looked at simultaneously.

6.3.2 Differences in the Analysed Dependencies

The DSMs have slight differences in their actual entries. Because the tools order their rows differently, their DSMs are reproduced in an alphabetical ordering in Figure 28. The differing entries are marked with a darkened background colour in both DSMs. The reasons for these differences are examined below.

The missing dependency in dtangler's DSM from package 2 to package 12 can be attributed to dtangler ignoring template dependencies. For example, consider the following method signature:

```
public PrintableResult(List<Failure> failures)
```

The dependency to Failure is not recognised by dtangler. This is an error in the dependency handling and is listed in dtangler's product backlog.

The reason for the missing dependency from package 13 to package 10 is not so clear. LDM's dependency details show that class Suite makes a class reference to Runner. When looking at the relevant class level DSM in dtangler, there is no dependency reported from Suite to Runner. When looking at the Suite source file the following line is quite an obvious reference to Runner:

```
Runner childRunner= Request.aClass(each).getRunner();
```

It is unclear why this particular dependency is ignored by dtangler; this requires further examination to be solved.

The remaining four differences in the DSM entries represent dependencies that dtangler found but LDM didn't. The most significant difference is when counting dependencies from package 8 to package 1: dtangler's dependency weight for this particular cell is 15 while LDM's is only 3. The missing dependencies come from Class objects of different types. For example, LDM doesn't recognise a dependency to Test if it is used in the following way:

```
return getAnnotatedMethods(Test.class);
```

The same applies to all the dependencies LDM missed in column 7.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1 org.junit				2			1	15					1
2 org.junit.experimental.results													
3 org.junit.experimental.theories				3	3								
4 org.junit.experimental.theories.internal			1										
5 org.junit.experimental.theories.suppliers													
6 org.junit.internal	1	1								1			
7 org.junit.internal.requests										5			
8 org.junit.internal.runners			4	2			7			1			13
9 org.junit.matchers	1												
10 org.junit.runner		3				2	14	10			4	5	1
11 org.junit.runner.manipulation							3	15		1			
12 org.junit.runner.notification		2				2	1	9		6			2
13 org.junit.runners							1						

	1	2	3	4	5	6	7	8	9	10	11	12	13
1 org.junit				2				3					1
2 org.junit.experimental.results													
3 org.junit.experimental.theories				3	3								
4 org.junit.experimental.theories.internal			1										
5 org.junit.experimental.theories.suppliers													
6 org.junit.internal	1	1								1			
7 org.junit.internal.requests										5			
8 org.junit.internal.runners			4	2			5			1			13
9 org.junit.matchers	1												
10 org.junit.runner		3				2	14	10			4	5	2
11 org.junit.runner.manipulation							3	15		1			
12 org.junit.runner.notification		3				2	1	9		6			2
13 org.junit.runners													

Figure 28. A comparison of DSM entries between dtangler (above) and LDM (below).

As the dependencies were analysed, an additional point for improvement was discovered in dtangler. As recalled, the DSMs in dtangler are not nested. The internal DSM of any entry that has a lower scope can be looked at by double-clicking on that cell. The problem with this was that it was time-consuming to distinguish between

dependencies that occur between the parent scope dependent and dependee, and dependencies that occur inside either parent. A typical scenario would be that the user wants to see exactly what classes cause a dependency between two packages. These dependencies should be somehow emphasised in the user interface. LDM doesn't have this problem because its information screen provides the necessary dependency details for any DSM entry.

A workaround for this problem in dtangler is to forbid the dependency on the parent scope. This will cause all dependencies violating that rule to be displayed with a yellow background in the DSM on the lower scope as well. This is not a proper solution because the need to highlight certain dependencies is not always an indication that these dependencies should be forbidden.

6.3.3 Displaying Cyclic Dependencies

The displaying of cyclic dependencies differs greatly between the tools. In dtangler, the aim is to make every cycle visible and give details of all the cycles in the violation window. LDM shows the cycles more discretely. They are not listed as violations or outlined in any other way than that there are entries above the diagonal. LDM uses cycles as a basis for grouping tangled items to the black squares.

To view detailed cycle information in LDM, the user has to generate a report of the analysed material. The cycle report of org.junit.* packages is shown in Figure 29. The cycles are listed as cycle groups, not as individual cycle paths.

The cycle groups in LDM list what row items are used in each black square after partitioning the DSM. For example, the largest square in the DSM (see Figure 27), contains rows org.junit.*, org.junit.internal, and org.junit.runner, and their sub-systems.

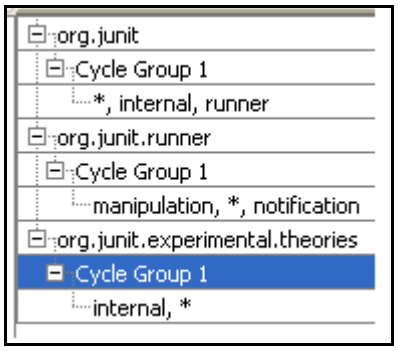


Figure 29. LDM cycle report.

The equivalence of LDM's report to dtangler's cycles is presented in Figure 30. Whereas LDM only displays the names of the participants to cycles, dtangler shows all cycles inside each cycle group.

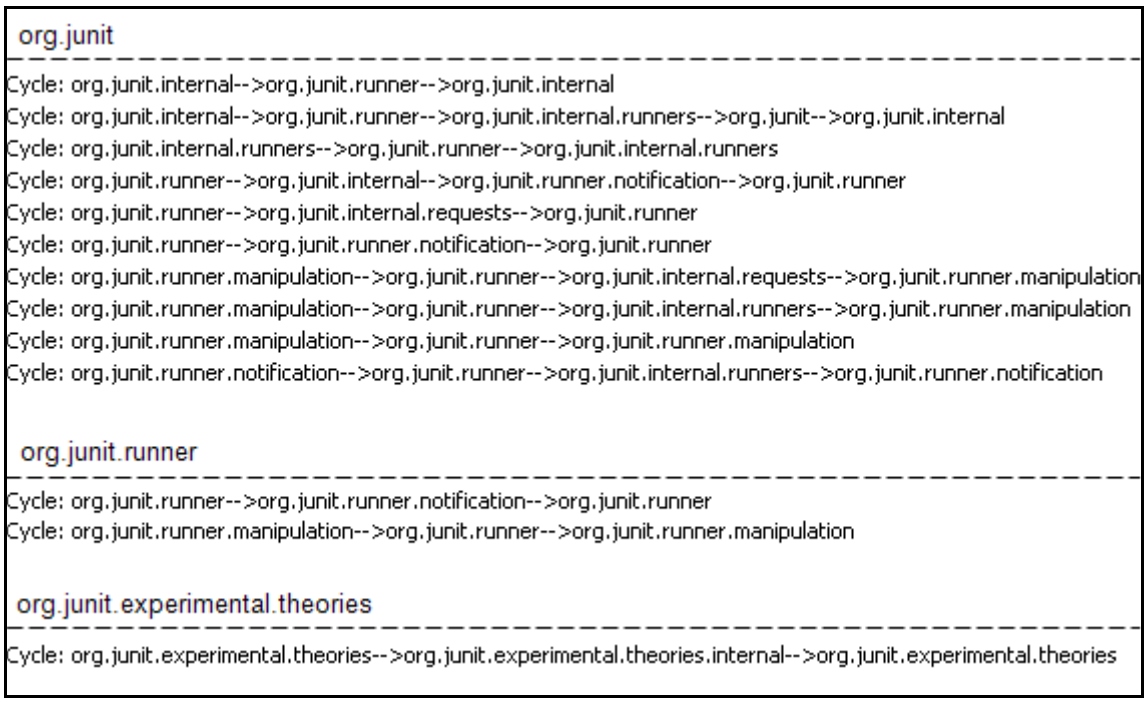


Figure 30. The cycle violations in dtangler sorted under LDM's cycle groups.

Figure 31 shows four cycles that aren't visible in LDM but are found by dtangler. The reason why LDM doesn't discover these cycles is because LDM misses the only dependency from package org.junit.internal.requests to package org.junit.runners. By adding this dependency manually, LDM's cycle report for org.junit will correctly include the org.junit.runners package.

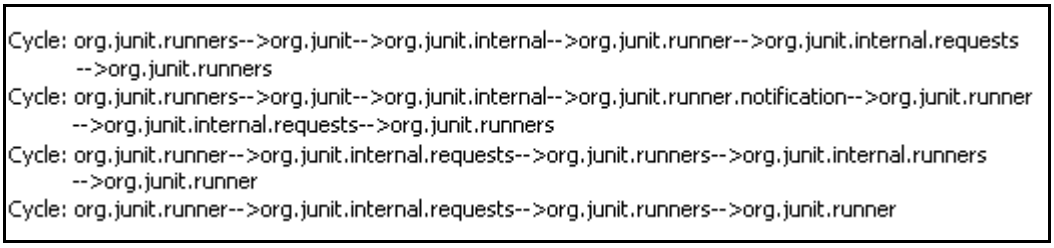


Figure 31. Additional cycles discovered by dtangler.

From these cycle reports, it is clear that dtangler provides more detailed cycle information than LDM. Where LDM only shows what items participate to cycles, dtangler also shows cycle paths and also makes note of cycles on lower scopes. The user can limit the cycle listing to any subset of DSM cells in dtangler by selecting them from the DSM and pressing the 'Selected violations' tab in the user interface.

6.3.4 Displaying Dependency Rules

LDM displays dependency rules as coloured triangles on the DSM cells, rows, and columns. Yellow colour is used to indicate that dependencies in this cell are forbidden, green is used for allowed dependencies. Rule violations are marked with a red colour. All of these indicators can be toggled on and off from the user interface. Rule violations are also listed textually.

An example of rule violations in LDM is shown in Figure 32. Here, a dependency rule has been entered that denies all dependencies from TheoryMethod to ParameterizedAssertionError. Both of these classes reside in the package org.junit.experimental.theories.internal. The user interface shows red triangles on the row names, column numbers, and on the violator cell. The Violations tab on the right also displays the violation textually.

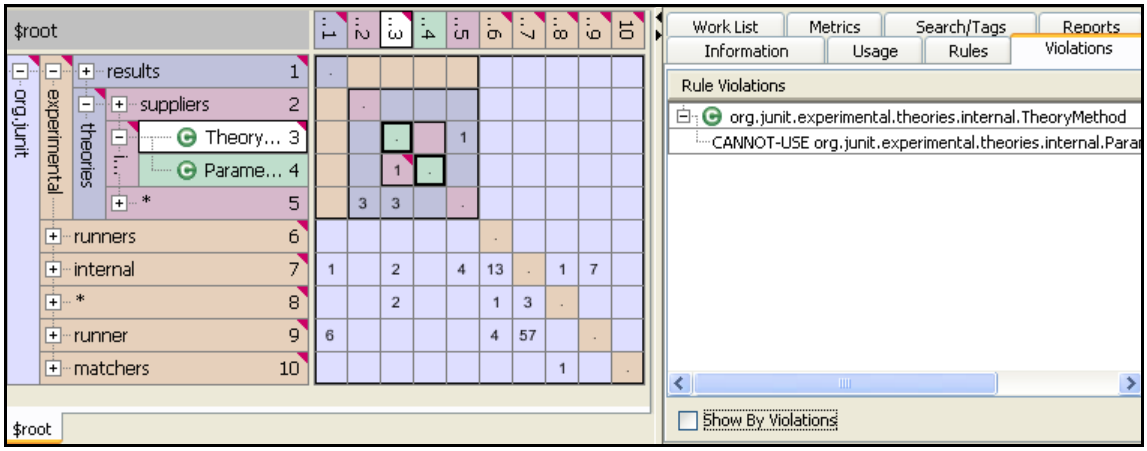


Figure 32. Displaying a forbidden dependency with LDM.

LDM considers dependencies to external libraries as rule violations by default. This is the cause of the red triangles not pointing to the org.junit.experimental.theories.internal package in the previous screenshot. These markers point to classes that reference classes outside the org.junit.* package structure, such as org.hamcrest.*. This conflicts a bit with the fact that all dependencies inside the DSM are allowed by default in LDM. It would make more sense if the existence of external dependencies was indicated in some other way than by just marking these dependencies forbidden; it is little confusing that there are rule violations before the user has even entered any rule definitions.

In dtangler, rule violations are marked with a yellow background colour on the DSM cells and rows. The violations are also listed textually. Figure 33 presents how the same rule violation is shown with dtangler. This screenshot is taken from the package scope, so the dependency violation is shown as a yellow background colour on row 5. If the user would zoom in to that specific package, any cells causing the dependency would be coloured yellow.

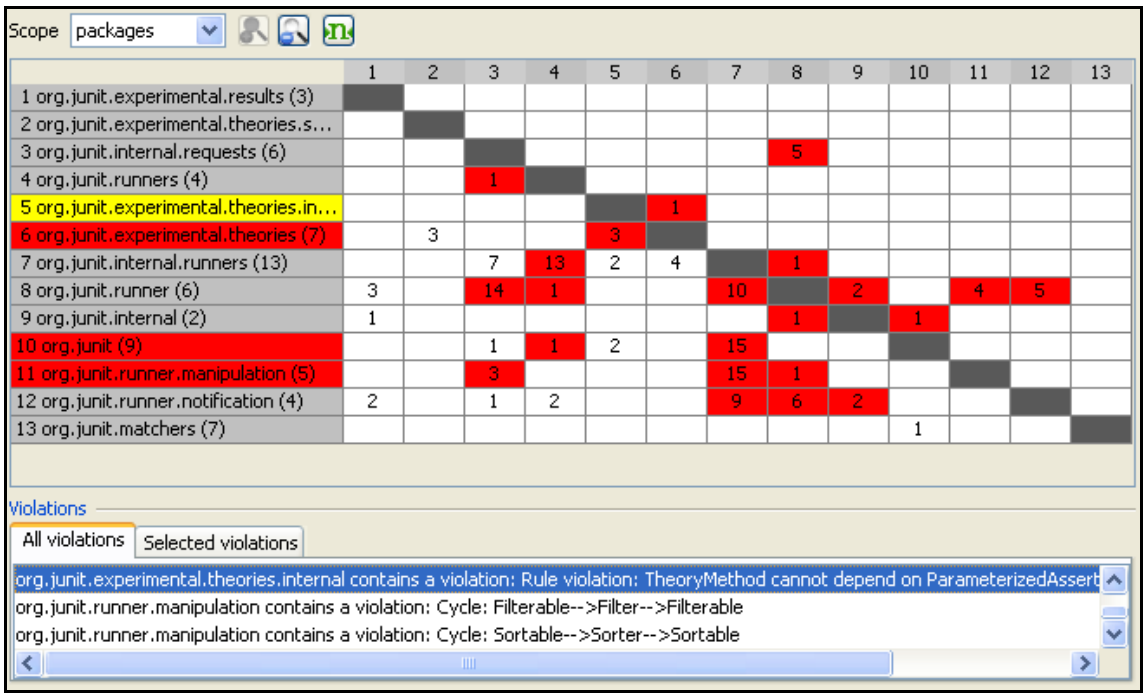


Figure 33. Displaying a forbidden dependency with dtangler.

There are no fundamental differences in the tools' methods for displaying rule violations. LDM does offer the chance of visualising rule definitions in the DSM with yellow and green triangles whereas in dtangler, only violated rules are highlighted in the DSM. Both tools support viewing and editing defined rules from a separate rule dialog window.

Another advantage of LDM in rule displaying is that all the markers on the DSM can be toggled on and off. The colour highlighting in dtangler, for both rules and cycles, is always on.

6.4 End User Experiences

The end users' thoughts of dtangler were collected with a web survey. A link to the questionnaire was posted on the dtangler web site. The goal was to find out what the users considered to be the strengths and weaknesses of the tool, and to discover how the tool is being used. The questionnaire used in this survey is presented in Appendix A.

Eleven end users replied to the survey, all of them being professional software developers. The majority of respondents (10 of 11) have used dtangler in their work, two respondents also in their own software projects.

Figure 34 describes how frequently dtangler has been used by the respondents. Six of the respondents use the tool quite rarely, less than once a month. However, it is encouraging that rest of the respondents have used the tool somewhat more regularly, either weekly or monthly.

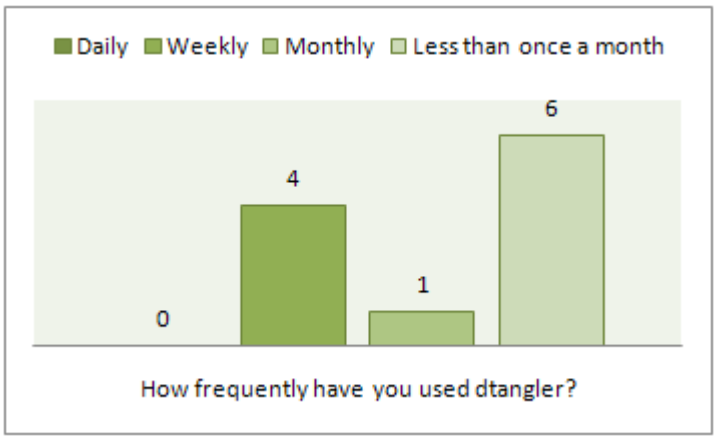


Figure 34. Respondents' frequency of using dtangler.

Dependency analysis tools have been associated mostly to refactoring and maintenance activities. Interestingly, in this survey the most common activity to use dtangler with was implementation (see Figure 35). This result would imply that dtangler has been successful in integrating with the implementation phase of software projects. Refactoring and maintenance activities ranked second and third. A couple of respondents have also used the tool in conjunction with software design and testing.

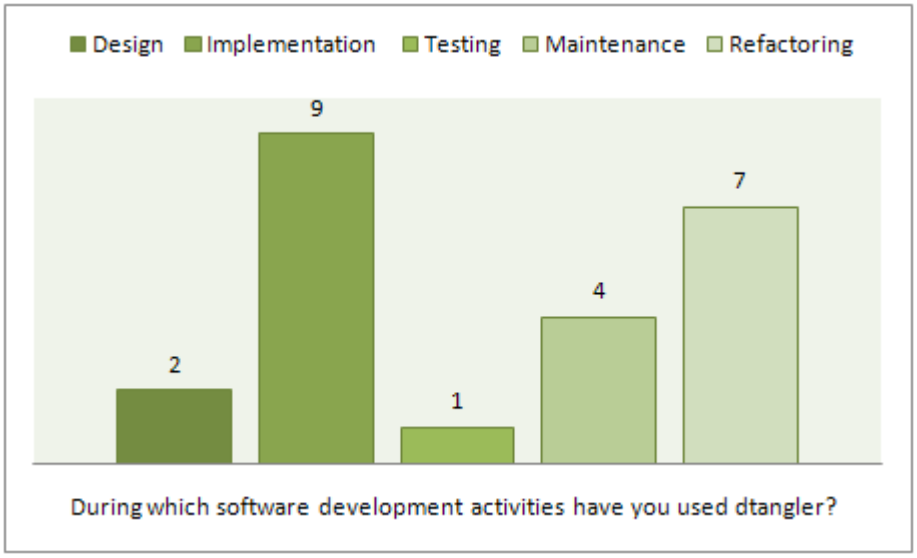


Figure 35. Respondents' use of dtangler in different software development activities.

The usefulness of dtangler's main features was examined. The average and standard deviation of the responses are presented in Table 8. The features gained average usefulness values from 4.3 to 3.1. The maximum value for usefulness was 5, the minimum was 1. The standard deviation of the answers ranged from 0.75 to 1.05. The conclusions that can be drawn from these are that all features were found somewhat useful and that the respondents' answers didn't vary drastically between one another.

Table 8. End users' view on the usefulness of dtangler's features, on a scale of 1 (not useful at all) to 5 (very useful).

	Java dependency analysis	Cycle and rule violation detection	Saving and loading configurations	DSM output on multiple detail levels	Dependency rule configuration
Average	4.3	4.2	3.9	3.6	3.1
Standard deviation	0.90	0.75	1.05	1.03	0.83

Java dependency analysis and cycle and rule violation detection were thought to be the most useful features, while configuring dependency rules was considered the least useful. There appears to be a slight conflict in these results: the second most useful feature is related to rule violation detection but the least useful feature is related to rule configuration. It could have been assumed that if the *detection* of rule violations is found useful, the *configuration* of the rules should also have been found useful, as the two are clearly connected.

One explanation for this is that users appreciated cycle detection more than rule violation detection; they were both asked about in the same question. Instead of considering cycle and rule violation as a single feature, this question should have been divided into two separate features in the questionnaire. Another reason for this somewhat conflicting result could be that configuring dependency rules is not relevant to every developer involved in the project: while they might find the identification of dependency violations important, they might never make their own dependency rules.

This part of the questionnaire would have benefited from further improvement. The questions in this section were of type "how useful is *feature X* in dtangler to you?" In addition separating different features more precisely, the question itself is a bit ambiguous. The reason for the usefulness or uselessness of any feature remains unclear: did the user think that the feature poorly implemented to start with, or was the feature something that the user wouldn't want to use even if it were properly implemented?

The final part of the questionnaire collected the end users' thoughts on dtangler's strengths, weaknesses, and areas of improvements with open questions.

The general ease of use and the existence of a graphical user interface were commended. The users found dtangler easy and fast to take into use. Frequent updates and distributing the software free of charge were also mentioned as dtangler's strengths. The general selection of features, and more specifically, the option of integrating dtangler into automated builds, was also liked.

The weaknesses of dtangler were related to efficiency problems and the lack of IDE integration. It was noted that dtangler cannot deal with large input materials efficiently, or in worst cases, at all. One of the problems with no IDE integration is that the exact source code causing a dependency cannot be accessed directly. Also, it was noted that using standalone tools requires more effort from the user.

The improvement suggestions given by the respondents were closely connected to the problems the respondents listed. The IDE plug-in and direct source code access were the most wanted improvements. Improving the performance of the tool when dealing with big software projects and adding the ability to analyse dependencies from input sources other than Java were also wanted.

6.5 Summary

In general, the core DSM features are the same in dtangler and LDM. Both tools offer DSM partitioning, dependency analysis on multiple detail levels, dependency rule specification and monitoring, and cycle detection. In many cases, LDM's functions are more customisable; for example, the user can filter the DSM, create virtual hierarchies to it, and toggle dependency rule markers on and off. A couple of features stand out that differentiate the tools. LDM includes direct refactoring from the DSM, whereas dtangler allows for dependency rule creation pre-emptively, during design time.

There were small differences in comparing the tools' dependency analysis results. The biggest difference is related to how much information the tools provide on cyclic dependencies. LDM's cycle information is minimal in comparison to dtangler's. The dependency rules function similarly in the tools even though LDM offers ways to customise their appearance on the DSM. When comparing the dependency analysis results between the tools, it was discovered that some forms of dependencies are missed by dtangler, some by LDM.

The end user experiences brought up some good points. It showed that the tool is being used by professional software developers to help them in their work, mostly during implementation activities. The end users most appreciated dtangler's usability, availability, and its selection of features.

The survey also gave new directions for issues that need to be addressed in the future, one of them being the need for dependency analysis of large projects. The fact that dtangler cannot handle large input material efficiently, or at all, is alarming. Software systems today are large, and the bigger the software gets, the more it needs tool support for analysing dependencies. Other wanted additions were an IDE plug-in and dependency analysis for languages other than Java.

7. Conclusions

The research problem of this thesis is answered in the following section, after which the general applicability of this research and its effects on practical software development are discussed. Finally, possible directions for future research on this field are given.

7.1 Answer to the Research Problem

The research problem that this thesis sought to answer was:

- How can software dependency analysis with DSMs be automated?

The problem is answered below by solving the sub-problems that were identified in Chapter 1.1. The application of the results to the development of dtangler is also outlined in each sub-chapter.

7.1.1 What Are Software Dependencies and How Can They Be Analysed?

The definition and categorisation of software dependencies varies in previous research. First of all, dependencies can be categorised according to their abstraction level: design, implementation, or cross-level dependencies. The focal point of this thesis was on implementation level dependencies.

Implementation level dependencies can be further categorised into static, dynamic and knowledge dependencies. Static dependencies are those that can be obtained from compiled source code; dynamic dependencies can include additional run-time dependencies; knowledge dependencies occur when an item indirectly affects another item.

The analysis of static dependencies is done by parsing binary files created by a compiler. Dynamic dependencies obviously need run-time analysis of the software being analysed. Knowledge dependencies can be analysed by a developer who thoroughly understands the conceptual relationships between the software modules.

Another way to define dependencies is to classify them as direct, transitive, or cyclic. Cyclic dependencies are thought to be the worst dependency type because they greatly reduce the understandability, testability and maintainability of software. They can effectively transform software into one big tangle of dependencies that has no cohesive modules with clear responsibilities. The abundance of dependencies of any kind is generally deemed an indication of degenerated software design that is in need of refactoring.

Finally, dependencies can be qualified by their significance: does the dependency occur on interface or on implementation level? Favouring interfaces over direct implementations is an efficient way of reducing the amount of change propagation.

All of the dependency analysis tools reviewed in the research analyse static dependencies, and this is also the case with dtangler. Due to their critical nature, cyclic dependencies have received special attention in dtangler. The tool displays them quite clearly and also provides information about exact cycle paths.

However, the last dependency classification method, implementation or interface dependencies, is not analysed by dtangler. It considers each dependency to be of equal significance. This is something that could be looked into as dtangler is developed further.

7.1.2 What Are DSMs and How Are They Produced?

Software dependencies can be displayed in a special matrix form, as DSMs. A DSM is a square matrix in which rows and columns represent the items whose dependencies are being modelled. Each entry in the DSM is an indication of a dependency *from* the column item *to* the row item. DSMs are particularly useful when there are a lot of items to analyse because traditionally used box-and-arrow style graphs do not scale well to big materials.

DSMs can be generated manually by analysing the dependencies and entering them to the matrix. However, the preferred method is to automate the DSM creation by using a specific dependency analysis tool that can output dependencies in a DSM form.

DSMs are not very useful until their rows have been ordered according to some dependency related criteria. Row ordering brings out the structure of the analysed dependencies and makes it easy to spot problem areas from the DSM. DSMs that model software dependencies are usually ordered with the partitioning algorithm. An ideal partitioned DSM dependency structure is lower triangular: all its entries are below the diagonal. Any entries above the diagonal are an indication of cyclic dependencies.

DSMs display coupling between any two items by showing the dependency weight in the DSM entry. Cohesion of a module can be analysed by looking at the partitioned DSM of the module: the module is highly cohesive if it is lower triangular.

The DSM ordering algorithm used in dtangler is a modification of the partitioning algorithm. The goal is the same in both algorithms: there should be no entries above the diagonal. But unlike in partitioning, these entries do not *necessarily* represent cycles. The rows are ordered by calculating the stability of each item and placing the least stable items upmost. This resembles how partitioning sorts items: items with no dependees are put to the top. The end result is mostly similar to partitioning but dtangler's sorting algorithm is not as forgiving: a software system without any cycles might still get entries above the diagonal as an indication of improper layering.

7.1.3 What Are the Requirements for a DSM Tool?

The requirements of a DSM tool can be derived from the answers to the two previous research problems. A DSM tool should be able to:

- Analyse software dependencies
- Order the DSM rows to produce a meaningful representation of the dependencies

These two requirements are fundamental – without them, you don't have a proper DSM tool. To survive against competition, additional features are almost certainly required. The features of commercial DSM tools were analysed, and are summarised below:

- DSMs can be presented on multiple detail levels
- The detail levels are nested to the same parent DSM
- Dependency details for each DSM entry are shown
- DSMs show dependency rule violations
- DSMs highlight cyclic dependencies
- The software can be refactored directly from the DSM

None of the commercial tools reviewed contained all of the features listed above. Out of these features, dtangler implements multiple detail level dependency analysis, dependency rules, and cyclic dependency highlighting.

7.1.4 What Are the Benefits and Problems of DSM Tools?

The general benefits of DSM tools are that they automate the dependency analysis and can display dependencies of large software systems well. They outline any problems in the software dependency structure by partitioning the DSM.

One of the problems with DSM tools is that reading DSMs requires a little additional effort from the user to get started; the way how DSMs display dependencies is quite different from the usual box-and-arrow style graphs. Another problem inherent to DSMs is that identifying transitive dependencies by looking at the matrix is not simple but requires scanning back and forth between the rows and columns.

The more specific benefits found in commercial DSM tools are related to their selection of features and were listed in the previous sub-chapter. The integration of DSMs with dependency rules and refactoring, as is done in Lattix LDM, adds a lot of value to the dependency analysis with DSMs. In effect, the addition of the dependency rules transforms a DSM tool from a dependency analysis tool to a dependency analysis and management tool. This opens up opportunities for integrating the tool closely with the software development process.

As with transitive dependencies, identifying cyclic dependency paths by looking at the DSM is difficult. This becomes increasingly difficult when the length of the dependency path grows. The inability to list the cyclic dependency path clearly to the user is one of the biggest problems that were found in the reviewed DSM tools.

The most obvious benefit dtangler has over the commercial tools is that it is open source. During the evaluation of dtangler, its features were compared to those of LDM, and it was discovered that dtangler's features are not as extensive LDM's. However, dtangler has are a couple of benefits over LDM: it outputs detailed cycle information and its dependency rules can be sketched out during the design phase. LDM reports what items take part to cycles but doesn't provide specific cycle details. The dependency rules in LDM cannot be specified until there is a DSM that contains the items to which the rules will be attached.

The dtangler end user survey revealed that the respondents liked dtangler's availability, usability and features the most. The respondents' thoughts on dtangler's problems consisted of missing IDE integration, no multiple input language support, and performance issues. These are all critical improvements. The addition of dependency analysis for multiple languages and IDE integration would attract a considerable amount

of new potential dtangler users. The efficiency improvement is also critical because currently, the lack of it prevents the analysis of large input materials and effectively cuts off many interested users.

7.2 Discussion

When comparing the tool constructed in this research to a commercial product, it should be noted that, without a doubt, the resources available to these projects differ greatly. The results are promising, considering that dtangler was developed by a small team that worked on this project part-time for half a year. The fact that dtangler can perform a lot of the same features as a commercial tool, free of charge, is a good incentive for companies to try the tool out and see if it is helpful in their software projects.

Even though dtangler has the core functionalities expected of a DSM dependency analysis tool, many features can be added and improvements made. DSM dependency analysis would be greatly enhanced if the tool would be made into an IDE plug-in. This would allow direct jump-to-code dependency details, and make dependency violations appear immediately in the IDE as soon as they are made. This would shorten the time span from creating a bad dependency to discovering and fixing it.

The evaluation of commercial DSM tools was done by the author. Even though care was taken in evaluating the tools and trying out their features, there is a possibility that some of the tools' functions were overlooked in the evaluation. The situation is a bit biased because the author naturally has thorough knowledge of dtangler's features. This should be taken into consideration when evaluating the reliability of the feature comparisons.

The DSM output comparison of LDM and dtangler doesn't suffer from bias because the evaluator only enters the same input material to the tools and highlights any anomalies between them; the tools do the actual dependency analysis without interference from the evaluator.

The end user survey balances the evaluation by giving voice to evaluators other than the author. However, the survey has its own limits, the most significant being that the amount of respondents was quite small and by no means enough to make any definite conclusions about dtangler. Also, the questionnaire should have been more carefully planned as some of the questions were ambiguous. On the other hand, these problems do not imply that the survey served no purpose. For example, the respondents' opinions about the pros and cons of the tool gave valuable information to its evaluation and pointers to further development.

7.3 Suggestions for Future Research

The future improvements to dtangler identified in this study brought up a whole new set of important research problems.

A very technical approach would be to investigate and improve the efficiency of the dependency analysis in dtangler. The efficiency bottleneck currently seems to be related to cycle detection, especially to finding the complete cycle path. One possible solution that could be looked into is applying the DSM partitioning algorithm to find cycle participants. This would restrict the search for cycle paths to the dependencies found on the upper triangular of the DSM.

The extraction of dependencies from input sources other than Java could also be studied. Two of the commercial tools analysed in this thesis, Structure101 and Lattix LDM also include a generic XML format for inputting dependencies between anything the user can imagine. In addition to built-in dependency support for various programming languages, the XML approach could also be researched.

This thesis focused on the core functions and implementation aspects of dtangler, which left a lot of uncovered ground to user interface related studies. The usability of the tool could be researched with user experiments.

Another completely different aspect would be to address the matter from a process point of view. The effects of dtangler on real-life software projects could be analysed by field studies. For example, does the tool initiate or encourage changes in the project's development process or in the behaviour of individual software developers? Is the quality of the finished software program better than the quality of previous, similar products?

References

Austin, S., Baldwin, A., & Newton, A. (1996). A data flow model to plan and manage the building design process. *Journal of Engineering Design, 7*(1), 3-25.

Baldwin, C., & Clark, K. (2000). *Design rules: Volume 1, the power of modularity*. Cambridge, Massachusetts: The MIT Press.

Bennett, K. (1995). Legacy systems: Coping with success. *IEEE Software, 12*(1), 19-23.

Bieman, J. M., & Kang, B. K. (1995). Cohesion and reuse in an object-oriented system. *Proceedings of the 1995 Symposium on Software Reusability,* Seattle, Washington, USA. 259-262.

Boehm, B. W., & Papaccio, P. N. (1988). Understanding and controlling software costs. *Software Engineering, IEEE Transactions on, 14*(10), 1462-1477.

Boehm, B., & Basili, V. R. (2001). Software defect reduction top 10 list. *IEEE Computer, 34*(1), 135-137.

Bohner, S. A. (2002). Software change impacts an evolving perspective. *Software Maintenance, 2002. Proceedings. International Conference on,* Montreal, Canada. 263-271.

Briand, L., Devanbu, P., & Melo, W. (1997). An investigation into coupling measures for C++. *Proceedings of the 19th International Conference on Software Engineering,* Boston, USA. 412-421.

Browning, T. R. (2001). Applying the design structure matrix to system decomposition and integration problems: A review and new directions. *Engineering Management, IEEE Transactions on, 48*(3), 292-306.

Cai, Y. (2006). Modularity in design: Formal modeling and automated analysis. (Doctor's Degree of Philosophy, University of Virginia).

Chen, K., & Rajich, V. (2001). RIPPLES: Tool for change in legacy software. *Software Maintenance, 2001. Proceedings. IEEE International Conference on,* Florence, Italy. 230-239.

Chidamber, S. R., & Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices, 26*(11), 197-211.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on, 20*(6), 476-493.

Cox, L., Delugach, H. S., & Skipper, D. (2001). Dependency analysis using conceptual graphs. *Proceedings of the 9th International Conference on Conceptual Structures*, Palo Alto, CA. 117-130.

de Souza, C. R. B. (2005). On the relationship between software dependencies and coordination: Field studies and tool support. (Doctor's Degree of Philosophy, University of California). Retrieved August 22, 2008, from http://www2.ufpa.br/cdesouza/pub/cdesouza-dissertation.pdf

Dhama, H. (1995). Quantitative models of cohesion and coupling in software. *The Journal of Systems & Software, 29*(1), 65-74.

Eder, J., Kappel, G., & Schrefl, M. (1992). Coupling and cohesion in object-oriented systems. *Conference on Information and Knowledge Management,* Baltimore, USA.

Eppinger, S. D. (1991). Model-based approaches to managing concurrent engineering. *Journal of Engineering Design, 2*(4), 283-290.

Fenton, N. (1994). Software measurement: A necessary scientific basis. *Software Engineering, IEEE Transactions on, 20*(3), 199-206.

Fowler, M. (1999). *Refactoring: Improving the design of existing code.* Addison-Wesley Professional.

Fowler, M. (2001). Reducing coupling. *IEEE Software, 18*(4), 102-104.

Gopal, A., Mukhopadhyay, T., & Krishnan, M. S. (2002). The role of software processes and communication in offshore software development. *45*(4), 193-200.

Grosser, D., Sahraoui, H. A., & Valtchev, P. (2003). An analogy-based approach for predicting design stability of java classes. *Software Metrics Symposium, 2003. Proceedings. Ninth International*, Sydney, Australia. 252-262.

Guo, J. (2002). Using category theory to model software component dependencies. *Engineering of Computer-Based Systems, 2002. Proceedings. Ninth Annual IEEE International Conference and Workshop on the*, Lund, Sweden. 185-192.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28*(1), 75-105.

Hitz, M., & Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. *Proceedings of the International Symposium on Applied Corporate Computing*, Monterrey, Mexico.

Huynh, S., & Cai, Y. (2007). An evolutionary approach to software modularity analysis. *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, Minneapolis, Minnesota, USA.

Jackson, D. (2004). Module dependences in software design. *Radical Innovations of Software and Systems Engineering in the Future: 9th International Workshop, RISSEF 2002*, Venice, Italy. 198-203.

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *IEEE Computer, 38*(9), 43-50.

Järvinen, P., & Järvinen, A. (2004). *Tutkimustyön metodeista* (Revised ed.). Tampere: Opinpajan kirja.

Jungmayr, S. (2002). Testability measurement and software dependencies. *Proceedings of the 12th International Workshop on Software Measurement*, Magdeburg, Germany.

Koch, A. S. (2004). *Agile software development: Evaluating the methods for your organization*. Boston, MA: Artech House.

Lakos, J. (1996). *Large-scale C++ software design*. Reading, Massachusetts: Addison-Wesley.

LaMantia, M. J., Cai, Y., MacCormack, A. D., & Rusnak, J. (2008). Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, Vancouver, BC, Canada. 83-92.

Li, W., & Henry, S. (1993). Maintenance metrics for the object oriented paradigm. *Software Metrics Symposium, 1993. Proceedings., First International*, Baltimore, Maryland, USA. 52-60.

Lindholm, T., & Yellin, F. (1999). *The Java™ virtual machine specification* (2nd ed.). Reading, Massachusetts: Addison-Wesley. Retrieved August 19th, 2008, from http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

Lopes, C. V., & Bajracharya, S. K. (2005). An analysis of modularity in aspect oriented design. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development,* Chicago. 15-26.

MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science, 52*(7), 1015-1030.

Mancoridis, S., Mitchell, B. S., Chen, Y., & Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on,* Oxford, UK. 50-59.

Martin, R. C. (1994). *Object oriented design quality metrics: An analysis of dependencies.* Retrieved May 20, 2008, from http://www.objectmentor.com/resources/articles/oodmetrc.pdf

Martin, R. C. (1996, December). Engineering notebook: Granularity. *C++ Report,* Retrieved May 20, 2008, from http://www.objectmentor.com/resources/articles/granularity.pdf

Martin, R. C. (1997, February). Engineering notebook: Stability. *C++ Report,* Retrieved May 20, 2008, from http://www.objectmentor.com/resources/articles/stability.pdf

Matos, P., Duarte, R., Cardim, I., & Borba, P. (2007). Using design structure matrices to assess modularity in aspect-oriented software product lines. *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques,* Minneapolis, Minnesota, USA.

Melton, H., & Tempero, E. (2006). Identifying refactoring opportunities by identifying dependency cycles. *Proceedings of the 29th Australasian Computer Science Conference,* Hobart, Australia. 35-41.

Melton, H., & Tempero, E. (2007). An empirical study of cycles among classes in java. *Empirical Software Engineering, 12*(4), 389-415.

Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. *Software Engineering, IEEE Transactions on, 27*(4), 364-380.

Nordberg III, M. E. (2001). Aspect-oriented dependency inversion. *OOPSLA Workshop on Advanced Separation of Concerns,* Tampa, Florida, USA.

Ohlsson, M. C., von Mayrhauser, A., McGuire, B., & Wohlin, C. (1999). Code decay analysis of legacy software through successive releases. *Aerospace Conference, 1999. Proceedings. 1999 IEEE,* Aspen, Colorado, USA. *5*. 69-81.

Oloufa, A. A., Hosni, Y. A., Fayez, M., & Axelsson, P. (2004). Using DSM for modeling information flow in construction design projects. *Civil Engineering and Environmental Systems, 21*(2), 105-125.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on, SE-5*(2), 128-138.

Pfleeger, S. L., & Bohner, S. A. (1990). A framework for software maintenance metrics. *Software Maintenance, 1990., Proceedings., Conference on,* San Diego, CA, USA. 320-327.

Rogers, J. L., Korte, J. J., & Bilardo, V. J. (2006). *Development of a genetic algorithm to automate clustering of a dependency structure matrix.* No. L-19157; NASA TM-2006-214279. NASA Langley Research Center. Retrieved June 18, 2008, from http://hdl.handle.net/2060/20060007869

Rusnak, J. (2005). The design structure analysis system: A tool to analyze software architecture. (Doctor's Degree of Philosophy, Harvard University).

Sangal, N., Jordan, E., Sinha, V., & Jackson, D. (2005). Using dependency models to manage complex software architecture. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications,* San Diego, CA, USA. 167-176.

Schwaber, K. (1995). Scrum development process. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95) Workshop on Business Object Design and Implementation,* Austin, Texas. Retrieved August 14, 2008, from http://jeffsutherland.com/oopsla/schwapub.pdf

Stevens, W., Myers, G., & Constantine, L. (1974). Structured design. *IBM Systems Journal, 13*(2), 115-139.

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

Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B. (2001). The structure and value of modularity in software design. *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Vienna, Austria. 99-108.

Takahashi, R., & Nakamura, Y. (1996). The effect of interface complexity on program error density. *Software Maintenance 1996, Proceedings., International Conference on*, Monterey, CA, USA. 77-86.

The Design Structure Matrix Web Site. (2008). *DSM partitioning.* Retrieved July 30, 2008, from http://www.dsmweb.org/content/view/33/26/

The Eclipse Foundation. (2004). *Eclipse public license*. Retrieved August 27, 2008, from http://www.eclipse.org/legal/eplfaq.php

Tuura, L. A. (2003). Ignominy: Tool for analysing software dependencies and for reducing complexity in large software systems. *Nuclear Instruments and Methods in Physics Research, 502*(2-3), 684-686.

Vieira, M., & Richardson, D. (2002). Analyzing dependencies in large component-based systems. *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, Edinburgh, Scotland, UK. 241-244.

Xiao, C., & Tzerpos, V. (2005). Software clustering based on dynamic dependencies. *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, Manchester, UK. 124-133.

Yang, H. Y., & Tempero, E. (2007). Indirect coupling as a criteria for modularity. *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, Minneapolis, Minnesota, USA.

Yassine, A. A. (2004). An introduction to modeling and analyzing complex product development processes using the design structure matrix (DSM) method. *Quaderni Di Management, 9.* Retrieved 18th February, 2008, from http://www.iese.uiuc.edu/pdlab/Papers/DSM-Tutorial.pdf

Appendix A. Web Questionnaire

dtangler user experience questionnaire

You can answer the questions in English or in Finnish.

What is your profession?

How frequently have you used dtangler?

- ○ Daily
- ○ Weekly
- ○ Monthly
- ○ Less than once a month

During which software development activities have you used dtangler?

- ☐ Design
- ☐ Implementation
- ☐ Testing
- ☐ Maintenance
- ☐ Refactoring

In what situations have you used dtangler?

- ☐ Work
- ☐ Other:

How useful is Java dependency analysis in dtangler to you?

dtangler can analyse dependencies from Java class and jar files.

	1	2	3	4	5	
not useful at all	○	○	○	○	○	very useful

How useful is DSM output on multiple detail levels in dtangler to you?
dtangler can display dependencies on file location, package, and class levels.

	1	2	3	4	5	
not useful at all	◯	◯	◯	◯	◯	very useful

How useful is dependency rule configuration in dtangler to you?

	1	2	3	4	5	
not useful at all	◯	◯	◯	◯	◯	very useful

How useful is cycle and rule violation detection in dtangler to you?

	1	2	3	4	5	
not useful at all	◯	◯	◯	◯	◯	very useful

How useful is saving and loading configurations in dtangler to you?

	1	2	3	4	5	
not useful at all	◯	◯	◯	◯	◯	very useful

What would you consider as dtangler's strengths?

What would you consider as dtangler's weaknesses?

In your opinion, how could dtangler be improved?

Submit