DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Reducing software complexity by hidden structure analysis

Methods to improve modularity and decrease ambiguity of a software system

OSCAR BJUHR

KLAS SEGELJAKT

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Abstract

Software systems can be represented as directed graphs where components are nodes and dependencies between components are edges. Improvement in system complexity and reduction of interference between development teams can be achieved by applying hidden structure analysis. However, since systems can contain thousands of dependencies, a concrete method for selecting which dependencies that are most beneficial to remove is needed. In this thesis two solutions to this problem are introduced; dominator- and cluster analysis.

Dominator analysis examines the cost/gain ratio of detaching individual components from a cyclic group. Cluster analysis finds the most beneficial subgroups to split in a cyclic group.

The aim of the methods is to reduce the size of cyclic groups, which are sets of co-dependent components. As a result, the system architecture will be less prone to propagating errors, caused by modifications of components. Both techniques derive from graph theory and data science but have not been applied to the area of hidden structures before.

A subsystem at Ericsson is used as a testing environment. Specific dependencies in the structure which might impede the development process have been discovered. The outcome of the thesis is four to-be scenarios of the system, displaying the effect of removing these dependencies.

The to-be scenarios show that the architecture can be significantly improved by removing few direct dependencies.

Keywords

DSM, VSM, Hidden Structure Analysis, Directed Graphs, Graph Theory, Data Science

Abstract

Mjukvarusystem kan representeras som riktade grafer där komponenter är noder och beroenden mellan komponenter är kanter. Förbättrad systemkomplexitet och minskad mängd störningar mellan utvecklingsteam kan åstadkommas genom att applicera teorin om gömda beroende. Eftersom system kan innehålla tusentals beroenden behövs en konkret metod för att hitta beroenden i systemet som är fördelaktiga att ta bort. I den här avhandlingen presenteras två lösningar till problemet; dominator- och klusteranalys.

Dominatoranalys undersöker kostnad/vinst ration av att ta bort individuella komponenter i systemet från en cyklisk grupp. Klusteranalys hittar de mest lönsamma delgrupperna att klyva isär i en cyklisk grupp.

Metodernas mål är att minska storleken på cykliska grupper. Cykliska grupper är uppsättningar av komponenter som är beroende av varandra. Som resultat blir systemarkitekturen mindre benägen till propagering av fel, orsakade av modifiering av komponenter. Båda metoderna härstammar från grafteori och datavetenskap men har inte applicerats på området kring gömda strukturer tidigare.

Ett subsystem på Ericsson användes som testmiljö. Specifika beroenden i strukturen som kan vara hämmande för utvecklingsprocessen har identifierats. Resultatet av avhandlingen är fyra potentiella framtidsscenarion av systemet som visualiserar effekten av att ta bort de funna beroendena.

Framtidsscenariona visar att arkitekturen kan förbättras markant genom att avlägsna ett fåtal direkta beroenden.

Nyckelord

DSM, VSM, Analys av Gömda Strukturer, Riktade Grafer, Grafteori, Datavetenskap

Table of Contents

1 Introduction

Complex software systems can be described as a composition of interacting components. Interactions occur due to use of features in other components and data exchanges. Transformation of a component will thus affect the components it interacts with.

The evolution of a software system is highly dependent on the level of interaction between its components. If the components are highly interactive, modifications within one part will have side effects in many other components. This might lead to ripple effects, propagating the changes to other components, which prohibits the system from rapid evolution.

If the components are more independent, and less interaction occurs between them, modification of one component has less of an impact on the system. Therefore, less adaptation is required by the other components to achieve a stable form. If the leap between stable forms is shorter, the evolution of a complex system will be more efficient (Simon, 1962).

The distance between stable forms is dependent on the level of modularity in the system. Modularity is the degree to which components in a system can be separated and recombined to create stable and working forms of the system. Higher modularity leads to a more flexible system (Schilling, 2000).

Modularity has long been known to affect the value of a system (Matthew J. LaMantia, Yuanfang Cai, Alan D. MacCormack, John Rusnak, 2008). Many studies have applied design structure matrix (DSM) to assess system's degree of modularity. The DSM is an adjacency matrix which captures direct interactions between components. In the DSM, direct dependencies are indicated by the value 1 in the cell located at the row of the source component and column of the target component (Eppinger & Browning, 2012). But this is a shallow estimation of the modularity. To achieve a more detailed assessment of the software architecture a visibility matrix (VSM) can be applied. The VSM is an adjacency matrix, generated from the DSM, which captures both direct and indirect dependencies. Hence, it gives clearer view of the underlying hidden structure of a system (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

Whilst the classification and display of the system architecture is clear, ways to improve the system and where the improvement efforts should be aimed are unclear.

Substituting or breaking dependencies have different impact on the system. By detaching components from the system, propagation of changes is avoided. However, some components might be heavily intertwined in the system and are therefore harder to detach. These components should not be targets for decoupling since they may require substantial effort to do so. More loosely coupled components are easier to detach but might not affect the system much.

Hence the improvement should be aimed at components that are relatively easy to decouple and have a great positive impact on the system if detached. The aim of this study is to develop an algorithm that finds these improvement points and estimates the gain from the improvements.

1.1 Background

Lack of modularity due to unclear architectural structures has been proven to impede the development of software systems (Giovanni, 1982). This is due to the fact that small changes may affect other components, requiring updates of their implementation. Changes made by the updates may propagate further (Robert Lagerström, Carliss Baldwin, Alan MacCormack, Stephan Aier, 2014).

Thus, an optimal chronological order for the completion of the components exists. Susbsystems that affect a large part of the system but are unaffected by other components should be implemented first. This grants the system a solid basis. Components that only depend on the basis, but affect a large part of the system, should then be prioritized.

A hierarchical structure is beneficial when determining the priority of development for different components. When components of higher priority are completed, changes in lower priority components should not propagate back and require updates of components with higher priority (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

Modularity is a feature that contains propagation of changes within a fixed set of components, called a module. Thus it enables a more hierarchical structure. Modules should appear to be consistent to outside sources whilst the actual implementation may be volatile. This can be enforced with clear design rules concerning input and output to the modules (Parnas, 1972).

These design rules should be implemented as early as possible to prevent excess costs due to expensive reconstructions of the system. If this is not done, or the structure deteriorates, a refactoring of the complete system may be required (Steven D. Eppinger, Daniel E. Whitney, Robert P. Smith, David A. Gebala, 1994).

A key metric for assessing the risk of propagation within a system is the propagation cost. Propagation cost intuitively indicates the average portion of the system that is affected by any modification of the system (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

1.2 Terminology

Dependency:	Link between components.
Hidden Dependency:	Indirect dependency between components due to direct links to intermediate components.
Hidden Structure:	Underlying architectural structure existing due to hidden dependencies.
Design structure matrix (DSM):	Matrix displaying direct dependencies.
Direct Fan-In (DFI):	Number of ingoing direct dependencies to a component.
Direct Fan-Out (DFO):	Number of outgoing direct dependencies to a component.
Density:	Proportion of the DSM which contains dependencies.
Visibility matrix (VSM):	Matrix displaying direct and indirect dependencies.
Visibility Fan-In (VFI):	Number of ingoing direct- and indirect dependencies to a component.
Visibility Fan-Out (VFO):	Number of outgoing direct- and indirect dependencies to a component.
Propagation Cost:	Proportion of the VSM which contains dependencies.
Cluster Cost:	Metric used to evaluate modularity.
Cyclic Group:	Set of components that are all dependent on each other.
The Core:	Largest cyclic group in a system.
Shared Component:	Component which other components in the system are highly dependent on.
Periphery Component:	Component which is independent from the system.
Control Component:	Component which are highly dependent on other components in the system.
Dominator:	Node which all paths to a specific node passes.
Post-Dominator:	Node which all paths from a specific node passes.
Cluster:	Subset of components in a cyclic group which are tightly connected.
To-Be Scenario:	Possible future scenario of a system obtained by altering dependencies.

1.3 Problem

One of the main concerns when designing a software system is to keep the propagation cost as low as possible. High propagation cost is a sign of excessive dependence between components. Thus, changing a component might cause a ripple effect which requires refactoring of pre-existing components (Robert Lagerström, Carliss Baldwin, Alan MacCormack, Stephan Aier, 2014).

Dependencies dictate how information flows through the system. The flow can become difficult to understand for system developers if there are numerous hidden dependencies. This can lead to unclear errors when one part of the system, that other parts indirectly depend on, is modified.

This study aims to answer the question: “How can the system be transformed in such a way that the propagation cost decreases without removing an excessive number of direct dependencies?”

1.4 Purpose

The purpose of this degree project is to reach a deeper understanding of system architectures. The ability to design and evaluate complex systems is valuable to large scale development projects. In regards to software projects, it is specifically important to make the right decisions concerning system design early on. Expensive refactoring may be required if wrong decisions are made.

As of now, the area of hidden dependency analysis is fairly unexplored. Future research will be needed and this report hopes to bring attention to the subject area. Companies can also take part of the proposed solutions and use it for their own ends.

Ericsson is one of the largest providers of ICT in the world. About 40 percent of global mobile traffic runs through networks provided by Ericsson (Ericsson, 2016). As the system size increases, problems can arise if modifications tend to propagate through the system.

Ericsson's development teams are globally distributed. Coordination between these development teams may be difficult and time consuming. Thus the risk for propagation of changes between modules developed by distant teams may delay and impede the development process.

Increasing the modularity would benefit the software development at Ericsson. The modules could be distributed to different geographical location. Propagation of changes could then be handled locally.

Higher modularity would also create a clearer successive order to which the development of components should follow in order to avoid the need for reconstruction (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

1.5 Benefits and Ethics

Ericsson is a world leading provider of ICT services. The services Ericsson provides have a great impact on the general quality of life. Defects caused by the system structure can be avoided if the system architecture becomes more comprehensible. This will have a positive effect on the general quality of life due to enhanced communication possibilities.

In the perspective of the IEEE code of ethics, seven out of the ten defined policies are affected by this project (IEEE, 2016). The policies and their connection to the project are:

1. to accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;

The analysis of dependencies in code can be used as a basis for making decisions in projects at Ericsson. By improving the system architecture, the risk for software defects which would negatively affect the communication abilities of the public would decrease.

2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;

Analysing the system architecture will contribute to a clearer view of the system. This may lead to fewer conflicts between development teams at Ericsson.

3. to be honest and realistic in stating claims or estimates based on available data

The analysis of dependencies in code can be used as a basis for making claims and estimates regarding the quality of code at Ericsson. It is therefore important that all possible delimitations of this study are made clear. Wrong decisions due to hidden delimitations must not happen.

4. to improve the understanding of technology; its appropriate application, and potential consequences

The project improves the understanding of technology by illustrating potential design flaws in a system's architecture. Potential software design solutions found in this study may extend the research within the area.

5. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations

The project can be used as a way to improve technical competence among system developers.

6. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others

Through the project, technical work such as code implementation can be offered criticism with architecture analysis. Sources used in the project should be clearly credited.

7. to avoid injuring others, their property, reputation, or employment by false or malicious action

Ericsson’s reputation could be affected by the outcome of this project. If a less error prone architecture can be obtained, with the help of this project, Ericsson’s brand name may be positively affected.

1.6 Sustainability

Sustainability can be split into three aspects; economic-, social- and environmental sustainability (Lozano, 2008).

1.6.1 Economic Sustainability

The thesis' aim is to improve the architectural structure of a software system to avoid propagation of changes. This will reduce the required development efforts and the risk for defects due to propagation of errors. The result will be a less expensive process for the software development without negatively affecting the resulting product.

1.6.2 Social Sustainability

Communication capabilities are central for democracies. If this thesis can give Ericsson's products a structure less prone to propagating errors, the communication and interconnection of populations might improve.

1.6.3 Environmental Sustainability

One goal of this thesis is to avoid interference between development teams. This might affect globally distributed development teams. Coordination between these teams might require business trips if errors tend to propagate between their software. Hence the result from this thesis might lead to a reduction of business trips and thereby less emissions.

1.7 Goal

The sub goals of the project are to:

- Construct a parser which extracts direct dependencies from source code.
- Find hidden dependencies in the software system.
- Propose changes in the code architecture.
- Use the hidden structure matrix in order to create to-be scenarios.
- Validate and rate the to-be scenarios with cost/gain ratio.

1.8 Method Evaluation

A DSM is used to capture direct dependencies in the system. The main advantages of a DSM approach are its conciseness, clean visualization, intuitive understanding, opportunity for analysis and flexibility.

Compared to many other network modelling techniques, such as flowcharts, the DSM stays concise as the number of components in the system increases. Components and dependencies are visualized intuitively and can quickly be introduced to newcomers. The model is very flexible as it allows for many alterations such as visualizing weight, importance and other dependency metrics. Since the DSM represents a directed graph, it opens up the possibility for applying graph theory (Eppinger & Browning, 2012). Furthermore, tools such as Excel and Matlab are well suited for working with the DSM format.

A VSM is used to visualize the hidden dependencies of the DSM. It has the same benefits as the DSM since it follows the same structure. The main advantage of using a VSM over a DSM for modelling a system is that it makes cyclic groups easier to identify. However, a DSM is required for creating a VSM.

To analyse the DSM and VSM two strategies were applied; dominators and cluster analysis. Dominators is a commonly used technique by compilers when optimizing loops in code (Appel, 2002). Cluster analysis is prevalent in data mining to find patterns in a large set of data (Brian S. Everitt, Sabine Landau, Morven Leese, Daniel Stahl, 2010). Both strategies offer their own approach to solve the problem presented in the thesis.

1.9 Delimitations

The thesis was conducted at Ericsson under a none disclosure-agreement. Sensitive information regarding the subsystem which was examined has been excluded from the report.

All dependencies in the system are considered to be equally important. In reality, some dependencies are stronger and are therefore more likely to propagate changes. Evaluating the strength of coupling within a system would require a larger time frame than assigned to this study due to two reasons:

1) Some forms of coupling, such as data access and procedure calls, are complicated to extract. The coupling's form is required to determine the strength of the dependency.
2) Previous work is unclear on how to handle weighted coupling. Thus development of a method for how this should affect the VSM and DSM and how it should be displayed is needed.

Therefore, this is exempted from this thesis and left for future studies.

This study also presumes that explicitly declared links are dependencies. I.e. unused imports are viewed as dependencies. This is due to the same reason as above, time constraint.

The analysis of the software architecture does not take into account dependencies that follow the design decisions made. Hence all dependencies are viewed as equally plausible to remove.

1.10 Outline

After the introduction, the report has seven chapters. Prior work, techniques, models and metrics which comprises the theoretical background of hidden structures are described in chapter 2. Methodology for how the theory in chapter 2 is applied to this project is described in chapter 3. Work which was performed in the project is explained throughout chapters 4, 5 and 6. How dependencies are defined in the system and how they are extracted is told in chapter 4. Processing of parser output into DSM and VSM formats is described in chapter 5. DSM and VSM analysis and creation of to-be-scenarios are specified in chapter 6. Results from chapter 4, 5 and 6 are presented in chapter 7. Finally, results are discussed in correlation to goals specified in the introduction and conclusions are drawn in chapter 8.

2 Hidden Structure Theory

Large software systems are dependent on the interactions between their components. This is an effect of the distributed and collaborative nature of software development. The ability to modify features without the need to update pre-existing code will reduce a system's technical debt (Miguel A. Fortuna, Juan A. Bonachela, Simon A. Levin, 2011). Technical debt is an estimate of the long term financial cost of maintaining and developing a software system (Edith Toma, Aybüke Aurum, Richard Vidgen, 2012). It can be evaluated by examining the hidden structures that exist due to hidden dependencies (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013) (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

Direct coupling between related components have shown to improve product quality. This is an effect of the increased interaction between developing teams when the coupling between them is explicitly declared. This is especially crucial early in the development process, when many design decisions are made. An increased amount of direct coupling has also shown to be time consuming. The development can even be stalled if there is too much direct coupling within a system (Steven D. Eppinger, Daniel E. Whitney, Robert P. Smith, David A. Gebala, 1994). Thus the evolution of a software system depends on the design decisions that are made in the beginning of a project. Systems with unclear design rules and an ambiguous architecture tend to evolve slower (Giovanni, 1982). Refactoring the system's structure is time-consuming and expensive. Therefore, strict design rules concerning the architecture should be implemented early in the development process (Parnas, 1972).

An example is the Mozilla web browser project. The first versions of Mozilla did not have a clear architectural structure. As a result, the development generated a large amount of defects, each requiring small updates. In turn some updates generated new defects. This was recognized as a problem by the developing team and the system was redesigned. The resulting architecture was less prone to propagating defects. Thus parallel development of separate components, without large side effects outside the component, was enabled (Alan MacCormack, John Rusnak, Carliss Baldwin, 2006).

2.1 Dependencies

Systems consist of dependencies between components. Dependencies can either be direct or hidden. Direct dependencies are explicitly declared links between components. Hidden dependencies are indirect links between seemingly non-interacting components. These links are caused by dependencies to intermediate components (Yu Zhifeng, V. Rajlich, 2001).

The structure of dependencies and components can be displayed as a directed graph, where nodes are components and edges are dependencies (Eppinger & Browning, 2012).

For each edge, the node which the edge starts from is called the origin of the dependence. The node that the edge points to is called the target.

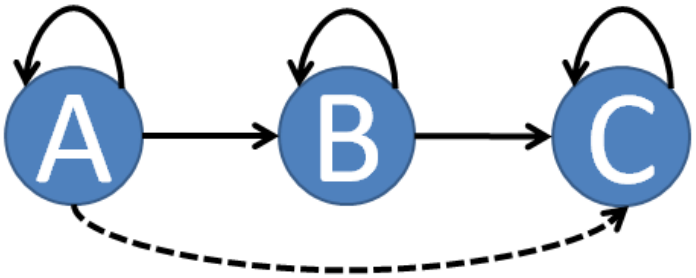


Figure 2-1 Dependencies between components A, B and C. Direct dependencies are visualized as arrows while indirect dependencies are visualized as arrows with dashes.

As an example, consider the graph in Figure 2-1 . A is directly dependent on B and B is directly dependent on C. This causes A to be indirectly dependent on C. Thus a hidden dependency exists, displayed with the dashed arrow. The edge between node A and node B has node A as the origin and node B as the target. In addition, every component in a system is considered to be directly dependent on itself.

2.1.1 Coupling

Coupling within a system is measured in two ways, strength and tightness.

Strength denotes a dependency's probability to propagate changes. Tightness measures the density of dependencies in a system or between a set of components.

The type of coupling can be studied to evaluate the strength of dependencies between components. An example is if a pair of components communicate through message passing. This is a weak coupling since the message passing acts like a barrier between components. They can be exchanged and modified without propagating the effects of the changes to the other component.

An example of a stronger coupling is a direct data dependency. This is when a component directly accesses a data structure in another component. If the component encapsulating the data structure is altered, affecting the data structure, the modification will propagate to the other component (Fenton & Melton, 1990).

2.2 Design Structure Matrix

A Design Structure Matrix (DSM) (Steward, 1981), also known as adjacency matrix or first-order matrix, is a modelling technique which captures direct dependencies between components in a complex system. Hence it is used as a representation of a directed graph. The technique has been used in a wide variety of industries ranging from pharmaceutical to aerospace engineering (Eppinger & Browning, 2012).

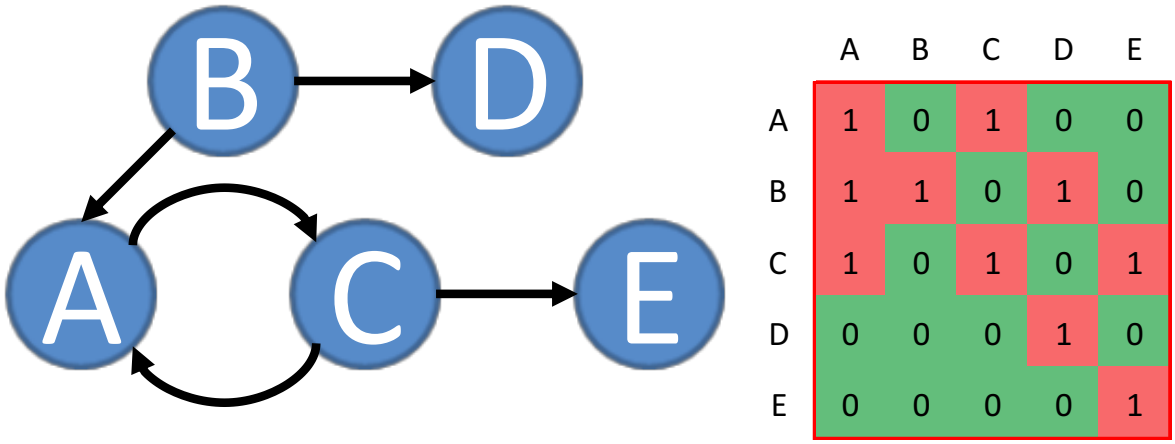


Figure 2-2 A directed graph and its DSM representation.

The DSM is visualized as a square *NxN* matrix where *N* is the number of components (Eppinger & Browning, 2012). A dependency from component *i* to component *j* is indicated in the matrix by a 1 in *i*:s row and *j*:s column. As an example, the dependency from B to A is indicated by a 1 in the row of B and column of A in Figure 2-2. All cells on the diagonal from the top left corner to the lower right corner are set to one. This is due the fact that all components are dependent on themselves (Steward, 1981).

The DSM in Figure 2-2 is a binary DSM which solely indicates the presence and absence of interactions between components. Thus the presence of a dependency is visualized with the value 1 and the absence is visualized with value 0. More complex forms of DSM:s can include the visualization of importance, impact or strength of each dependency. Non-binary values, symbols, shadings and colours can be used to visualize these attributes (Eppinger & Browning, 2012).

2.3 Visibility Matrix

A Visibility Matrix (VSM) is a modelling technique which extends the DSM technique. It captures both direct- and hidden dependencies (Eppinger & Browning, 2012).

VSM:s can come in different formats. In Figure 2-3, two different VSM types of the system in Figure 2-2 are displayed. The cells in the binary VSM only illustrate the absence or presence of hidden and direct dependencies. This can be altered to show the depth of each dependency as shown in the VSM on the right. The depth of a dependency is equal to the shortest path from the source node to the target node in the directed graph representing the system. As an example, node B has an indirect dependency with depth 3 to node E in Figure 2-3. This correlates to the fact that three edges need to be traversed to get from node B to node E in Figure 2-2.

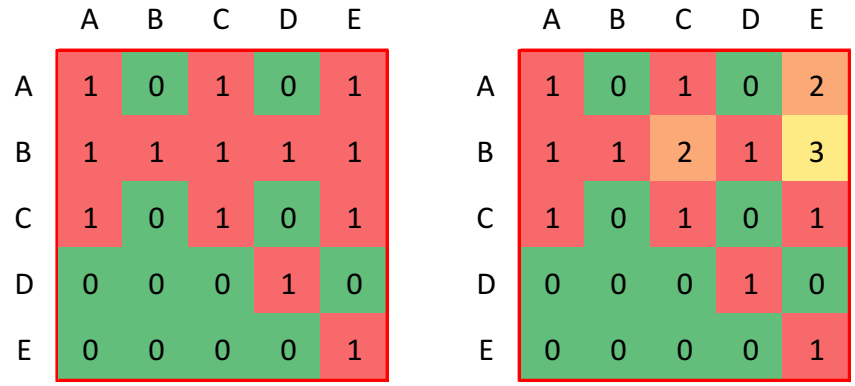


Figure 2-3 Binary VSM and depth VSM, both generated from the DSM in Figure 2-2.

Generating a binary VSM is a destructive process since direct dependencies are mixed with hidden dependencies. Therefore, the DSM cannot be reproduced from the binary VSM. On the other hand, the depth VSM is not destructive because it shows which dependencies are direct (Eppinger & Browning, 2012).

2.4 Classification of Components

Each component has an effect on the system depending on its coupling. Some components are more crucial to the complete system, whilst others are more independent (Michael L. Tushman, Lori Rosenkopf, 1992). The hidden structures can be examined to distinguish each component's role in the system. The classifications that exist are; core-, shared-, control- and periphery components (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

2.4.1 Core Components

One of the central hidden structures in the architecture is cyclic groups. These are a set of components that, either directly or indirectly, depend on all other components in the set (Johann Peter Murmann, Michael L. Tushman, 1997). Changes in any component in the cyclic group may require updating all other components in the group.

The largest cyclic group is called the Core, since it has a high coupling to the system. Changes in these components have large effects on the system (Michael L. Tushman, Lori Rosenkopf, 1992).

While strong interdependence within cyclic groups is beneficial, each cyclic group should strive for independence from components outside the cyclic group. This makes the system more modular (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

2.4.2 Shared Components

Other components that affect the evolution of a system are components with high pleiotropy. These are called shared components. High pleiotropy indicates that many other components depend on the component. Hence changes in this component may propagate into a large part of the system. This is an impeding feature and tends to prohibit development of the component. Greater repercussion need to be taken during modification of the component since it may have large side effects in the system (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). These components should be implemented early to enhance the development process (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

2.4.3 Control Components

Control components depend on a large part of the system, but have a small number of components that are dependent on them. These are highly affected by changes in the system, but changes in them do not propagate into a large part of the system (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). Control components should be implemented last, since they are highly affected by changes in the rest of the system. There is a higher risk that they need to be modified, due to changes in other components, if they are implemented early (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

2.4.4 Periphery Components

Periphery components are components that are separate from the system. They have a small amount of dependencies to and from the system. Changes in these components have a small probability of propagating and affecting the rest of the system. Likewise, changes in the system have a small chance of affecting the periphery components (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). Hence development of these components can be scheduled independently from the rest of the system (Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016).

2.5 Modular Architectures

Systems are in general categorized into two types of architectures; integral architectures and modular architectures. Integral architecture implies that the components in the system are tightly coupled. As a result, the system architecture becomes complex. Modular architecture seeks to reduce the complexity by decoupling components (Ulrich, 1995) (Sosa, et al., 2007).

A modular system is dependent on well-defined inputs and outputs of each module. This will make the intended purpose for each module clear (Parnas, 1972). These design rules could be implemented with interfaces and hiding the volatile parts that tend to change within modules. Thereby modules will appear more constant and reliable to sources outside the module (Matthew J. LaMantia, Yuanfang Cai, Alan D. MacCormack, John Rusnak, 2008). To achieve this, the system structure need to be of a modular architecture.

2.6 Techniques to Enhance Modularity

The general approach to improve a system's architecture is to make it more modular. Modularity can be achieved using different methods, i.e. decomposition, aggregation or the splitting operator (Eppinger & Browning, 2012) (Matthew J. LaMantia, Yuanfang Cai, Alan D. MacCormack, John Rusnak, 2008).

2.6.1 Further Decomposition

Components in a DSM may be aggregations of smaller components. *Decomposition* of these components may result in a less coupled system. The interactions within the system become clearer by breaking down components. It leads to a lower level of abstraction (Eppinger & Browning, 2012).

2.6.2 Aggregation

Aggregation, as opposed to decomposition, suggests that components can be merged. By merging components, the level of abstraction becomes higher. It may be a source

of errors since issues are hidden instead of being revealed (Eppinger & Browning, 2012). However, updates to correct errors within the components would be less visible to outside sources. Hence, the probability for propagation of modifications would decrease (Parnas, 1972).

2.6.3 The Splitting Operator

To enhance modularity, direct dependencies between components can be exchanged with dependencies to an intermediate interface. This is a way of decoupling components and breaking cyclic groups that is called the *Splitting* operator.

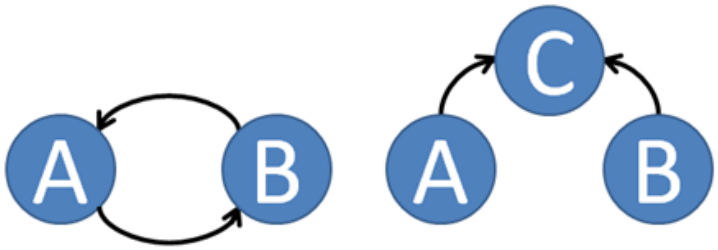


Figure 2-4 Breaking up a cycle by adding a new intermediate interface component.

It is executed by finding a set of components with direct dependencies to each other (A and B). Then a new interface (C) is introduced to the system. The interface captures the properties of the components in the set. The dependencies between the components are replaced with links to the interface. Thus the modules become more independent and can be modified and exchanged without affecting each other (Matthew J. LaMantia, Yuanfang Cai, Alan D. MacCormack, John Rusnak, 2008).

3 Methods for Displaying, Evaluating and Improving System Architectures

Grasping the overall structure of a complex software system is a complicated task. It might require a substantial period of time before a true picture of the structure is obtained. In order to make the system architecture more accessible and comprehensible to developers some visual models can be used, such as a DSM and a VSM. Some techniques, such as sorting and restructuring, can be applied to the models to further enhance the unambiguity of the models.

3.1 DSM Methods

To facilitate analysis of the DSM, the data that it contains has to be restructured. The types of data that need to be taken into consideration are the number of components, number of present dependencies and the position of those dependencies.

Through this data, three units of measurement are established. The units of measurement for the DSM are the Direct Fan-In (DFI), Direct Fan-Out (DFO) and density. DFI is the number of direct dependencies to a component. DFO is the number of direct dependencies from a component. Density is the number of present direct dependencies relative to the maximum number of possible direct dependencies.

3.1.1 DFI and DFO Values

DFI and DFO are natural number values that are measured by computing the DSM column and row sum respectively. Both values denote how tightly a component is directly connected to the rest of the system. DFI represents the number of components that directly depend on a specific component. DFO is the number of components a specific component directly depends on (Robert Lagerström, Carliss Baldwin, Alan MacCormack, Stephan Aier, 2014). An example of calculating the DFI and DFO of a component can be seen in Figure 3-1.

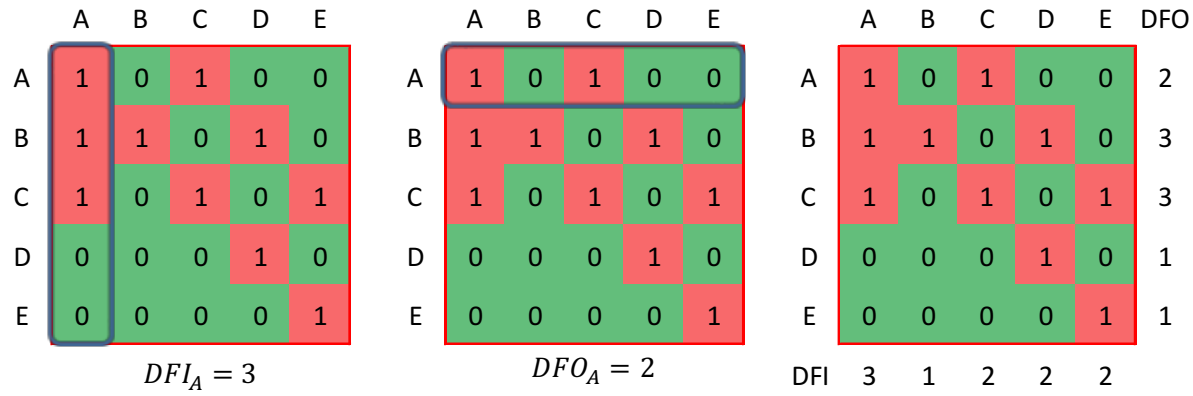


Figure 3-1 The DFI and DFO of component A are calculated by counting the number of direct dependencies in its row and column respectively.

3.1.2 Density

The density of a DSM is a decimal value representing the number of dependencies in a system in relation to the system size.

Density is mathematically defined as:

$$Density = \frac{\sum_{i=1}^{N} DFI_i}{N^2} = \frac{\sum_{i=1}^{N} DFO_i}{N^2}$$

Where N is the number of components in the system (Eppinger & Browning, 2012).

3.2 VSM Methods

Like the DSM, the VSM should also be restructured. The four units of measurement for the VSM are the Visibility Fan-In (VFI), Visibility Fan-Out (VFO), propagation cost and cluster cost.

In addition, the VSM needs to be sorted to reveal cyclic groups and hidden structures.

3.2.1 Transitive Closure

The VSM is generated by calculating the transitive closure of the DSM. Transitive closure can be computed in various ways. The most common methods for calculating it are with matrix multiplication or by applying Warshall's algorithm (Floyd, 1962). Both algorithms are computationally intensive with time complexity $O(N^3)$ where $N = |V|$ and V is the number of vertices (Ray, 2012). There are however more efficient algorithms such as Arlazov's with $O(N^3 \log N)$. Moreover, Fischer and Meyer have concluded that the problem can be solved in $\log_2 N$ multiplications of NxN matrices (Munro, 1971).

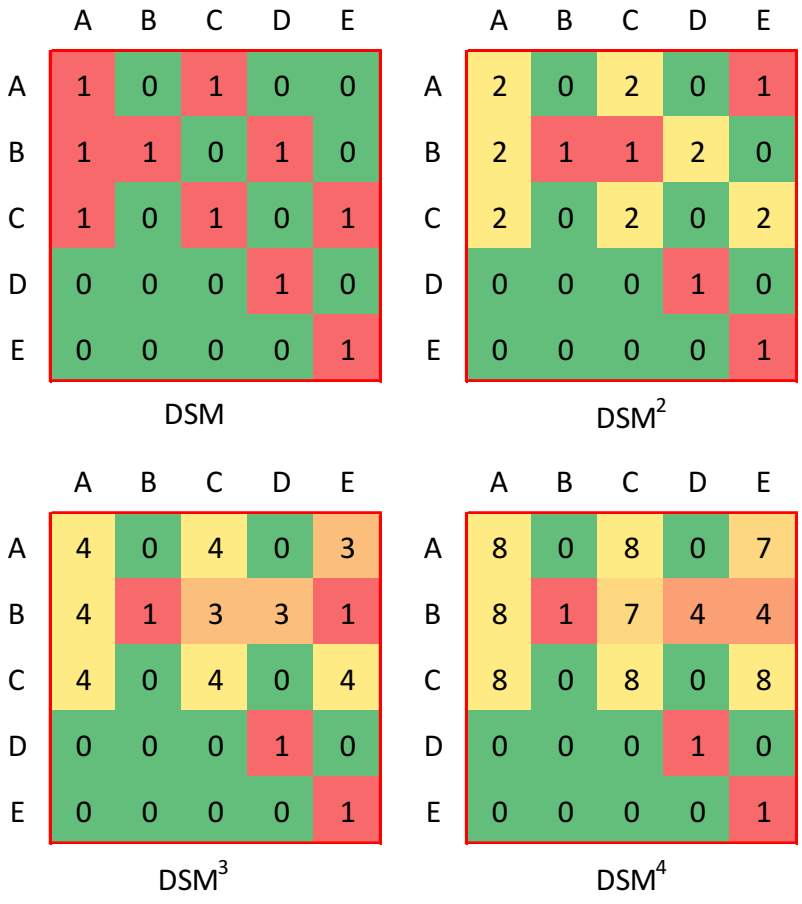


Figure 3-2 The transitive closure of a DSM through matrix multiplication.

In Figure 3-2 the transitive closure is computed by raising the DSM to power four. Dependencies can at most have depth N-1. To guarantee that all indirect dependencies are found the DSM^{N-1} needs to be computed. However, the algorithm can converge much earlier. Once the algorithm converges all hidden dependencies have been found and no further matrix multiplications are needed.

3.2.2 VFI and VFO Values

The VFI and VFO values extend the DFI and DFO. They count both direct and hidden dependencies. An example of calculating the VFI and VFO of a component can be seen in Figure 3-1.

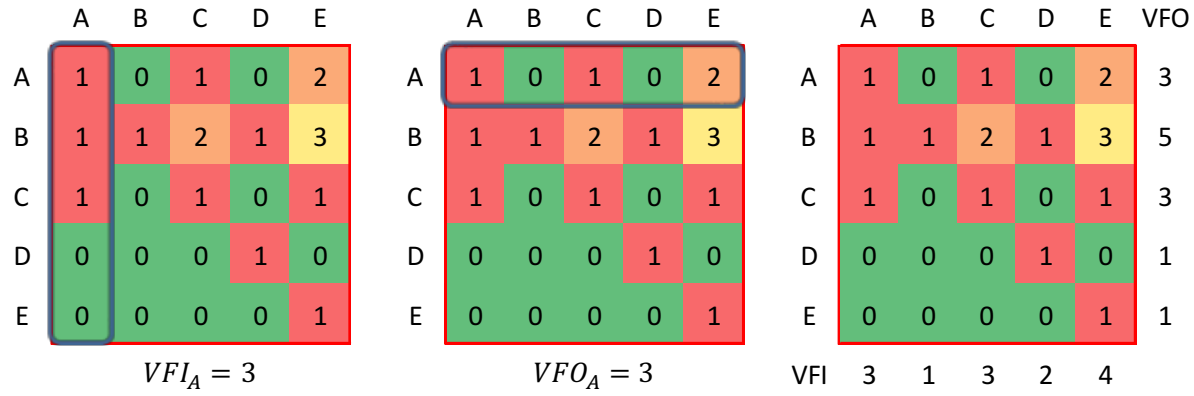


Figure 3-3 The VFI and VFO of component A are calculated by counting the number of direct and indirect dependencies in its row and column respectively.

3.2.3 Propagation Cost

Propagation cost denotes the average percentage of the system that will be affected by modifying a randomly selected component (Alan MacCormack, John Rusnak, Carliss Baldwin, 2006). A high propagation cost infers a high coupling within the system, both direct and indirect. This tends to create and propagate errors, due to the architectural structure, that are hard to detect (Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Julien Delange, Marco González, Philippe Kruchten, 2013).

Propagation cost is mathematically defined as:

$$Propagation\ cost = \frac{\sum_{i=1}^{N} VFI_i}{N^2} = \frac{\sum_{i=1}^{N} VFO_i}{N^2}$$

Where N is the number of components in the system (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). The VSM in Figure 2-3 has room for 25 dependencies, but only 13 dependencies are present. This causes the propagation cost to become 13/25 = 52%.

3.2.4 Cluster Cost

Propagation cost assumes that each dependency, both hidden and direct, have the same cost penalty for the technical debt. No matter the length of the path to the dependent component the value of the dependency is assumed to be the same. Cluster cost is an effort to evaluate the actual cost of dependencies.

The standard method for calculating is done by assigning different values for different dependencies. Dependencies within clusters are less expensive than dependencies to

components in other clusters. Dependencies to shared components are the least expensive dependencies (Alan MacCormack, John Rusnak, Carliss Baldwin, 2006).

3.2.5 Finding Cyclic Groups

All members of a cyclic group have the same VFI and VFO. Members of cyclic groups will therefore appear next to one another in the VSM if it is sorted after VFI or VFO.

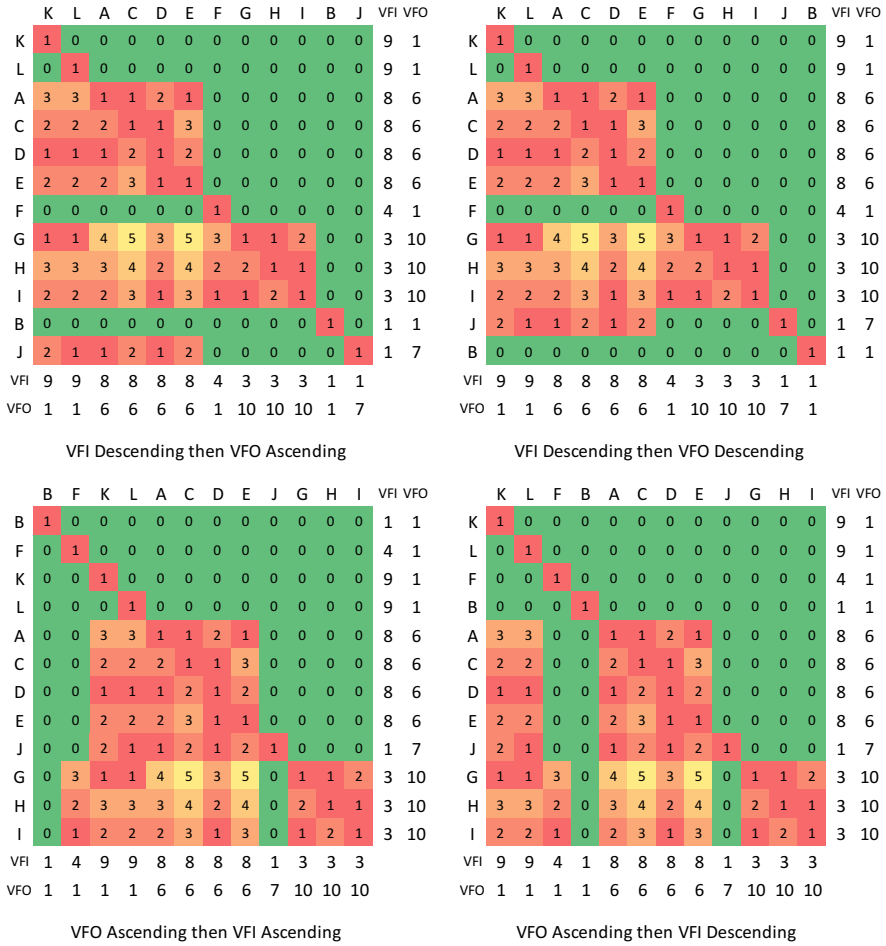


Figure 3-4 All possible sorting orders for the same VSM. The top right order 1.VFI Descending, 2. VFO Ascending order is used in the project.

Components are first sorted after their VFI in descending order and then VFO in ascending order. Thus if two components have the same VFI, they are sorted after VFO. The reason behind this specific sorting order is purely subjective. Components with many incoming dependencies are placed at the top of the matrix while those with many outgoing dependencies will be placed at the bottom. This reinforces the concept that information flows downwards in the VSM, resembling a water fall. All dependencies, except for those in cyclic groups, will be placed beneath the diagonal (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). An issue exists with this sorting; cyclic groups with the same VFI and VFO can be entangled. Hence the VSM might become cluttered. This is unlikely, especially if the system is larger. However, a solution for this problem is proposed in Appendix D.

3.2.6 Method for Classifying Components

To isolate the subsets of similar components, each system component is classified after its attributes. The classification is defined as:

- Core components are components that belong to the largest cyclic group. All components in a Core have the same VFI and VFO, represented as VFI_C and VFO_C.
- Shared components are components with $VFI \geq VFI_C$ and $VFO < VFO_C$.
- Peripheral components are components with $VFI < VFI_C$ and $VFO < VFO_C$.
- Control components are components with $VFI < VFI_C$ and $VFO \geq VFO_C$ (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

3.3 Architectural Classification

The system architecture can be evaluated and classified by examining the hidden structures. This can give an understanding of the system's complexity.

There are four system architectural patterns; Hierarchical, Multi-Core, Borderline Core-Periphery and Core-Periphery. Architectures are determined through comparing the system size and the Core size. The architectural classification scheme uses three threshold percentage values.

- Size threshold – Categorizes systems after their Core size which can either be small or large in context of the system size.
- Sensitivity threshold – Allows systems to be classified as "borderline" if the Core size is within close range of the size threshold.
- Dominance threshold – Draws a line between systems where the Core is much larger than the second largest cyclic group and systems where they are similarly sized.

The size, sensitivity and dominance are set to 5%, 1% and 50% respectively (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). The process of classifying a system is described in Figure 3-5.

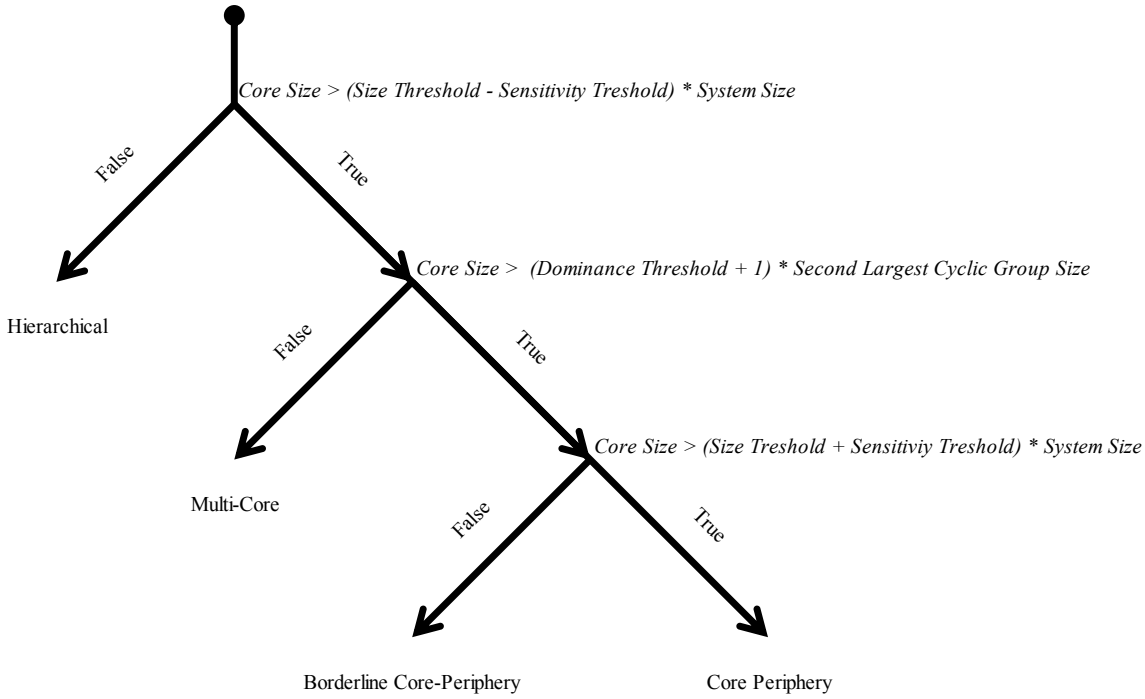


Figure 3-5 Flowchart describing the process of classifying a system's architecture.

A system has a hierarchical architecture if the number of components of the largest cyclic group is less than 4% of the number of components in the whole system.

If the largest cyclic group encapsulates more than 4% of the system, but is less than 50% larger than the second largest cyclic group, the system has a Multi-Core structure.

If none of the above conditions are met and the largest cyclic group is between 4% and 6% the system is described as Borderline Core-Periphery.

If the largest cyclic group contains more than 6% of the system, it is classified as a Core-Periphery architecture.

In general terms, systems should pursue a hierarchical architecture as it contributes to modularity (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

3.4 Cyclic Group Analysis

The architectural group within a system which tend to be beneficial to spend improvements efforts on is the Core. Redesigning the Core can improve the development process, as seen in the case of Mozilla. A smaller Core decreases propagation cost and system complexity (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013). To decrease the Core size, components need to be decoupled from it by removing or exchanging direct dependencies i.e. by using the *Splitting Operator, Decomposition* or *Aggregation*.

Decoupling a component is done by either breaking all direct dependencies to or from the other components in the Core. This is due to the fact that a cyclic path can only exist if there is a direct path to a node from the Core and from the node to the Core.

To find the most beneficial points to improve in the Core the gain of decoupling each node should be calculated.

The gain is calculated by the number of nodes within the Core that will be detached by breaking a set of edges. In order to calculate this, dominators and post-dominators are found for each node within the Core.

3.4.1 Dominators and Post-Dominators

Dominator is a technique used in graph theory. It is commonly used by compilers to examine control flow graphs in order to optimize loops. Node A dominates node B if all paths to B pass though A. Inherently, all nodes dominate themselves and the graph must contain a start node.

Post-domination is another technique of interest. It evaluates the opposite flow of direction and requires an end node instead of a start node. Node A post-dominates node B if all paths from B pass through A. The start- and end nodes are required in order to construct a dominator tree (Appel, 2002).

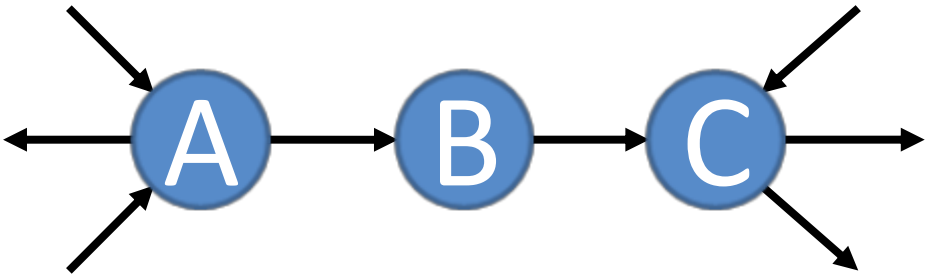


Figure 3-6 A directed cyclic graph with nodes A, B and C.

In Figure 3-6, node A dominates node B since all paths to B pass through A. Node C post-dominates node B since all paths from B pass through C. Node B does not dominate or post-dominate node A or C since there are paths to and from A and C which do not pass through node B.

Dominators are valuable when examining a cyclic set of nodes in a direct graph. A cyclic set only contains nodes that can reach all other nodes in the set. All edges to and from nodes outside the cyclic set are ignored.

All included edges go to a node within the cyclic set and all nodes within the set have a directed path to all other nodes in the cyclic set. Hence there is always a path back to the origin node from the target node for all included edges. Therefore, all included edges are part of at least one cyclic path.

Finding dominators and post-dominators in this set will thus lead to finding which nodes all cyclic paths leading to and from a node pass through. Detaching the dominator node from the cyclic group will thus break all cyclic paths to the dominated nodes. The same applies to the post-dominated nodes.

The theory behind dominators and post-dominators is commonly applied to control flow graphs and requires a start- and end node. Control flow graphs are used to represent all possible execution paths of programs, which have clear start- and end nodes. However, in the case of DSM graphs, the control flow is instead the flow by which changes propagate. Since changes can begin to propagate from any component in the system, no start- or end node exist. This is not a problem since the use of dominators and post-dominators in this thesis only include finding which nodes all paths to and from each node have to pass. The method for calculating this is still applicable without a start- and end node.

3.4.2 Finding Clusters in Cyclic Groups

A cyclic group may consist of several tight intertwined subgroups with a few edges between them acting like bridges. These groups and bridges would be beneficial to highlight in order to decouple and split the cyclic group.

Cluster analysis is used to solve this problem. It is a method for restructuring a large set of data into sets called clusters. The clusters are composed of components with similar attributes (Brian S. Everitt, Sabine Landau, Morven Leese, Daniel Stahl, 2010).

Hence cluster analysis can be applied to the cyclic groups in the VSM in order to display distinct intertwined subgroups. This could be done by restructuring the VSM in a way which the components that are closely intertwined are placed alongside each other. The direct links between the components in the subgroup will then create a clear square around the diagonal in the VSM. Direct dependencies acting as bridges between subgroups will be placed far from the diagonal.

3.4.3 Evaluation of Improvement

The propagation cost before and after refactoring is a crucial evaluation criterion. This is because propagation cost denotes the amount of coupling, direct and indirect, within the system. High propagation cost has been proven to be an impeding feature (Alan MacCormack, John Rusnak, Carliss Baldwin, 2006).

The cluster cost indicates the overall cost of the dependencies. A high cluster cost indicates a high number of deeper indirect dependencies. Deeper hidden dependencies are usually harder to comprehend by developer and tend to be error prone (Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Julien Delange, Marco González, Philippe Kruchten, 2013). Thereby, the cluster cost before and after a refactoring should be studied. The cluster cost is estimated as the accumulated depth of all hidden- and direct dependencies.

The Core size as a percentage of the system size should be studied. Decreasing the Core size is important to obtain a more hierarchical system, which in turn makes it easier for developers to comprehend the system architecture (Carliss Baldwin, Alan MacCormack, John Rusnak, 2013).

4 Dependency Extraction from the Source Code

Two types of dependencies are extracted: unit dependencies and block dependencies. Unit dependencies exist between software-, interface- and library unit directories in the system. Block dependencies occur at a higher abstraction level than unit dependencies.

All dependencies are converted to exist between directories of a given depth instead of between files. The DSM will grow immensely in size if this is not done.

4.1 Identifying Dependencies

Dependencies can be declared in two different ways:

- Regular source code imports, where the path to the target file is explicitly declared in the source file.
- Dependencies to units and paths in configuration files.

The configuration files contain the declarations of all xml units and which xml units that are required in order to build each unit. Dependencies between xml units should be converted to dependencies between folders. The origin of each unit is set to the folder where the unit is declared.

4.2 Extracting Dependencies

Dependencies are extracted using a parser written in bash script. The parser goes through a number of steps to extract all dependencies.

As output, the parser writes the found dependencies to three files.

- Units.txt - Dependencies between units.
- Blocks.txt - Dependencies between blocks.
- Dependencies.txt - Information on how to locate the source of each dependency.

4.3 Dependencies in .c, .h, .sig, .cpp, .java and .asm Files

The first step is to find all units by searching the repository for directories with names that contain the substring "swu", "ifu" or "lu".

The combination of block folder name and the underlying unit folder name are unique for each unit directory's path. Include paths that contain both the block directory name (block name) and the unit directory name (unit name), in the form "block name/unit name", are searched for in all .c, .h, .sig, .cpp, .java and .asm files. This will catch both absolute- and relative paths from other block directories. The gcc pre-processor is used to exclude comments in the source code. This worked since all comments follow C syntax.

If the pattern "block name/unit name" is found, a dependency is declared. This dependency is between the directory where the pattern is located and the folder identified by the "block name/unit name" pattern.

Relative paths to each unit directory are searched for in the files. It is done by searching for a relative path to the unit directory name, in the form of "../unit name", within the same block directory.

4.4 Dependencies in Configuration Files

An expression for the stream editor *sed* was implemented to remove comments and descriptions in configuration files. It ignores everything that is stated between "<*!*—" and "-->", which are the start and end symbol for a comment. Everything between the "<*desc*" and "<*desc*>" is also ignored since the tags indicate a description in the configuration files.

Paths and relative paths are then searched for in the uncommented source code, as in the previous section.

Then the declarations for the xml units are found. A declaration is discovered if a "</*interface*", "<*library*" or "<*process*" tag is located between a "<*swu name*=".*">" and "</*swu*>" tag. The directory where the declaration is found and the name of the xml unit, which is the string between the quotation marks in the "<*swu name*=".*">" tag, is saved together in a file. Each xml unit has a unique name.

All "<*swu name*".*"" tags in the system are then located. The xml unit names are saved together with the directory where the tags are found. The xml unit name together with the directory it is declared in are then obtained from the file generated in the previous step. The xml unit name is then replaced with the directory where it was declared. This represented a dependency from the directory where the xml unit is referenced to the directory where the xml unit is declared.

4.5 Verification of Dependencies

A tool was implemented to display the exact formatting of each dependency in the source files, together with the exact declaration if it is a dependency to a xml unit.

It is used to manually verify correctness of dependencies that the parser discovered. The size of the codebase that is examined in this thesis made it difficult to manually verify all dependencies. Instead, samples from the result are verified.

The results were verified with system experts at Ericsson. They validated that the components are classified correctly and that the VSM displays a realistic model of the system.

The verifier is also used to ensure that the parser finds all dependencies. The dependencies between two files are first examined manually. Then the parser is used to extract dependencies between the files. The two results are compared to ensure that the parser finds all dependencies between the files.

5 DSM and VSM Processing

To analyse the dependencies, they first need to be processed into a proper format. In general terms, the process was divided into three steps; 1) reformat dependencies from CSV to DSM, 2) compute the VSM from DSM, 3) generate a DSM with cyclic group sorting. It is carried out using an application in Excel implemented with Visual Basic. All of the matrices generated in the process are illustrated in Appendix A.

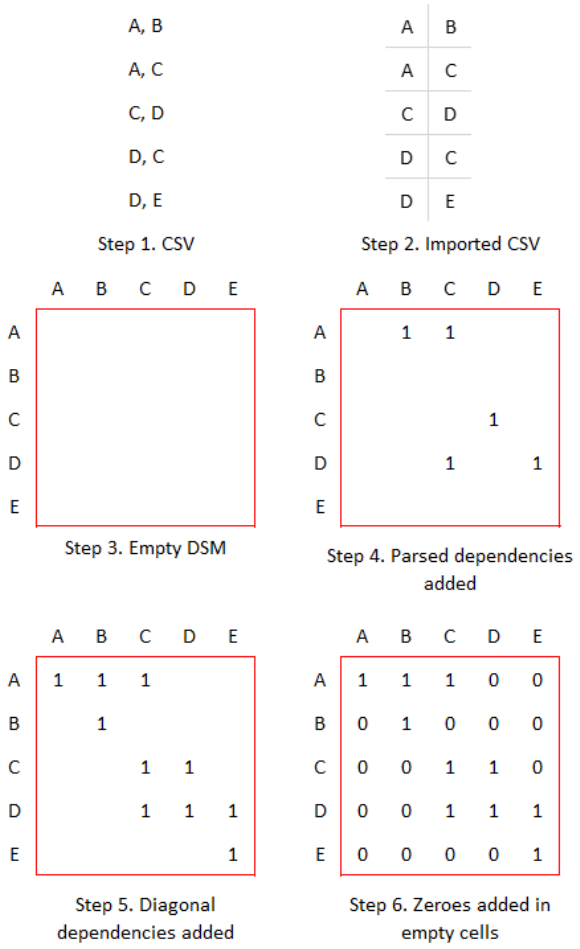


Figure 5-1 Steps of generating a DSM.

5.1 DSM from CSV

Excel provides basic functionality for parsing text files through delimitation. By setting the delimiter to ',' Excel can import data from the CSV file into a worksheet. An empty DSM is constructed where each unique component found in the data was assigned a row and column. The value 1 is then put in cells located by indexing the DSM with the source row and target column of each dependency. Afterwards each cell on the downward diagonal is assigned the value 1. Empty cells are assigned the value 0. The complete procedure is illustrated in Figure 5-1.

5.2 VSM from DSM

VSM transformation is accomplished through iteratively applying Excel's native matrix multiplication functionality. Each iteration is compared to its prior to check if

any new dependencies are introduced. If a new dependency is introduced, it is assigned the number of the iteration. If there were no new dependencies, it means that all indirect dependencies have been found. As a result, the VSM is fully generated.

VFI values and VFO values are then calculated by summarizing the number of dependencies per column and row respectively.

To reveal the cyclic groups, the VSM is sorted after its VFI in descending order and VFO in ascending order. Since the order needs to be preserved between sorts, it is crucial that the sort is stable. Excel's native sorting algorithm is stable and is thereby used.

5.3 Cyclic Group Sorted VSM

The VSM is then sorted after classification groups. Firstly, an algorithm locates the cyclic groups of the VSM by searching for mirrored dependencies down the diagonal. The cyclic groups are then sorted after size and the largest cyclic group is declared as the Core. VFIc and VFOc are set to be the VFI and VFO of the Core.

All components except for those in the Core and cyclic groups are assigned a value which indicates which classification group it belongs to. The components of the VSM are then sorted in the classification group order 1) Shared, 2) Core, 3) Cyclic Groups, 4) Periphery and 5) Control.

5.4 Calculating Architectural Metrics

When the VSM is completed, five metrics are calculated; propagation cost, cluster cost, Core size, second largest cyclic group size and size of the different classification groups. All sizes are calculated as a proportion of the complete system. The size of the Core and the second largest cyclic group are used to classify the architectural structure of the system. All of the metrics are calculated as described in chapter 3.

6 Analyzing the DSM and VSM

Components can be detached from the Core by breaking or exchanging dependencies. By doing this the Core size shrinks, propagation cost decreases and the system architecture improves.

Some components are easier to detach from the Core and have a greater impact on the Core size if detached. The generated DSM and VSM are used to calculate and display these components as key points for structural improvement.

6.1 Finding Dominated Components

The algorithm starts by finding the nodes which each node is dominated by, called the dominators. This is done to calculate the nodes each node dominates in a cyclic group. The function is recursive and only include edges and components within the cyclic group. All found dominators for each node are saved in a list. In the beginning, each node's list of dominators only contains themselves.

The algorithm then iterates over each node individually. The common dominators of all immediate predecessors, nodes that have a direct link to the target node, are found. This is due to the fact that all paths to a node have to pass through at least one of its immediate predecessors. Hence a component that dominate all immediate predecessors of a node also dominate the target node. If new dominators are found, which have not been detected in previous iterations, they are added to the list of dominators for the dominated node.

When the dominators of the last node have been calculated, the algorithm checks if any new dominators have been discovered. In that case the algorithm restarts, iterating over all nodes once more. All dominators have been found if no new dominators are added during an iteration. Then the algorithm will stop.

A new list is constructed for each node. All nodes which a target component is dominating are added to the target component's list of dominated components. This is calculated by finding in which lists from the previous iterations the target node occurs.

To calculate post-dominators, the same algorithm is executed with a small alteration. The common dominators of all the predecessors to the target node are not used. Instead the common post-dominators of all the targets immediate successors, nodes which the target node has a link to, are calculated.

The union of the dominated and post-dominated nodes are calculated and stored in a list. The resulting list contain all components that will be detached from the cyclic group if the target node is decoupled.

6.2 Finding Crucial Dependencies in a Cyclic Group

Components with a low number of direct dependencies to or from other components within the cyclic group are easy to detach. Decoupling them require breaking few dependencies, either all to or all from the cyclic group. These components are ideal to decouple if they dominate a large number of components. By detaching these components, all nodes they dominate are detached as well.

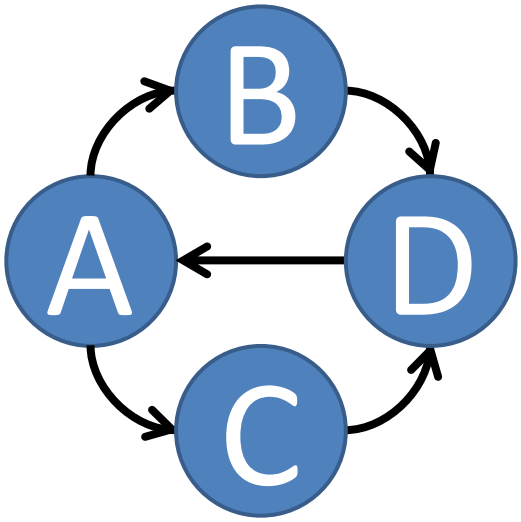


Figure 6-1 A directed graph displaying a cyclic group composed of four components.

In Figure 6-1 node B and node C does not dominate D. Node A dominates node B, C and D. Node D dominates all nodes. All nodes dominate themselves. Hence node A and node D dominate four nodes while node B and node C only dominate themselves.

All nodes can be decoupled by removing one edge. Therefore, either node A or D should be decoupled. The easiest way to detach node A is by removing its ingoing edges. Meanwhile, decoupling node D is most efficiently achieved by removing all of its outgoing edges. Thereby, the edge from D to A is the most beneficial to remove.

An algorithm is implemented where the average amount of components that are detached from the cyclic group per removed direct dependency is calculated.

$$FI\ Gain = \frac{Dominate}{DFI_{CG}}$$

$$FO\ Gain = \frac{Dominate}{DFO_{CG}}$$

DFI_{cg} and DFO_{cg} is the number of the direct dependencies, fan-in and fan-out, within the cyclic group for each element. *Dominate* is the number of nodes in the union of dominated and post-dominated nodes.

The cyclic group sorted VSM matrix is copied to a new matrix and all values in the cyclic group are set to zero.

All cells in a component's column within the cyclic group are set to its FI gain if the previous value is lower than the FI gain. The same is done with FO gain, but for cells in the component's row. Pseudo code of the algorithm:

```
For i from firstComponent to lastComponent {
        FoGain = calculateFoGain()
        FiGain = calculateFiGain()
        for j from firstComponent to lastComponent {
                if(cell[i][j] < FiGain)
                        cell[i][j] = FiGain
                if(cell[j][i] < FoGain)
                        cell[j][i] = FoGain
        }
}
```

FO and FI gain display the number of components that will be detached from the cyclic group per removed dependency. However, to guarantee that all dominated nodes are detached, all dependencies in the highlighted row or column need to be removed. Thereby, every direct dependency in a row or column with a relatively high value is recommended to break.

6.3 Finding Subgroups in Cyclic Groups

The VSM is restructured to display possible subgroups in each cyclic group. This is done by calculating a value for each component, based on the dependencies which exist to and from the component. The cyclic group is then sorted after the value.

After the restructuring is done, the possible subgroups are found and highlighted.

6.3.1 Sorting the VSM

Each component has a unique ID, which correspond to its current position in the cyclic group. The average ID of all dependencies to and from each component is calculated.

This value is calculated by adding all the ID:s for the targets of all outgoing dependencies and the origins of all ingoing dependencies for a component. The sum is divided by the component's number of ingoing and outgoing dependencies, DFI_{CG} and DFO_{CG} respectively. Dependencies to or from components outside the cyclic group are ignored.

The cyclic group is sorted after this value. Each component is given a new ID based on the placement in the cyclic group after the sort. This is done repeatedly N times or until the algorithm have converged, where N is the number of components in the cyclic group.

6.3.2 Finding and Highlighting Subgroups

A brute force algorithm was implemented to highlight subgroups. The algorithm calculates the number of hidden- and direct dependencies that are removed by removing a subset of the direct dependencies.

This is done by counting all direct dependencies to and from a subgroup and other components in the cyclic group, DFI_g and DFO_g respectively. This is due to the fact that the subgroup would be detached if all direct dependencies in one direction, between it and the rest of the cyclic group, is removed. Either all to or all from the subgroup. Cyclic paths between the subgroups can only exist if there are edges in both directions between the subgroups.

The area in the subgroup's rows or columns, that is not indicating dependencies within the subgroup, is calculated. It is used to examine the amount of dependencies that exist, both direct and indirect, due to direct dependencies to or from the subgroup.

The area is then divided by the DFI_g and DFO_g. The value indicates the amount of hidden dependencies that exist, on average, per direct dependency either to or from the cyclic group. A copy of the VSM is generated and the value of all cells in the cyclic groups are set to zero.

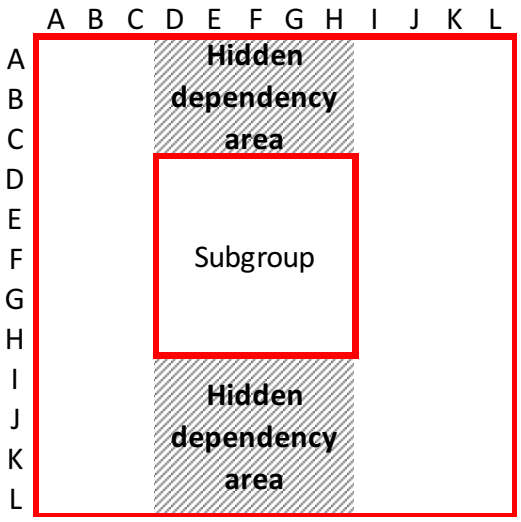


Figure 6-2 Displaying the area corresponding to the amount of external hidden dependencies to or from a subgroup.

The max value of the DFI_g and DFO_g is calculated. All cells in the subgroup are set to the negative representation of that value if it is greater than the absolute value in each cell in the subgroup.

All cells in the subgroup's rows, but outside its columns, are set to DFI_g if the absolute value of its previous value is smaller. The same is done for the subgroup's rows and DFO_g.

Pseudo code of the algorithm:

```
For each subgroup {
        FoGain = calculateSubgroupFoGain()
        FiGain = calculateSubgroupFiGain()
        MaxGain = MaxOF(FoGain, FiGain)
        If (subgroup.allCellsSmallerThan(MaxGain)) {
                For i from firstComponentInSubgroup to lastComponentInSubgroup {
                        For j from 1 to firstComponentInSubgroup - 1 {
                                if(absolute(cell[i][j]) < FiGain)
                                        cell[i][j] = FiGain
                                if(absolute(cell[j][i]) < FoGain)
                                        cell[j][i] = FoGain
                        }
                        for j from firstComponentInSubgroup to lastComponentInSubgroup {
                                cell[i][j] = -MaxGain
                        }
                        for j from lastComponentInSubgroup + 1 to LastComponentInCylicGroup {
                                if(absolute(cell[i][j]) < FiGain)
                                        cell[i][j] = FiGain
                                if(absolute(cell[j][i]) < FoGain)
                                        cell[j][i] = FoGain
                        }
                }
        }
}
```

Hence, areas with high values display the most beneficial set of direct dependencies to remove in order to split the cyclic group. Negative values indicate sets of components forming the most distinct subgroups. Notably, disjointing the subgroups do not ensure that they would form cyclic groups. The only guarantee is that no cyclic paths will exist between the subgroups. Thus, the subgroups will not be part of the same cyclic group after the split.

6.3.3 Finding all Connected Subgroups

All possible connected subgroups are examined with this algorithm. This is done by two nestled loops, the outer is set to the start component of the subgroup and the inner to the end component. The outer loop iterates from one to *N*, where *N* is the amount of components in the cyclic group. The inner loop iterates from the outer loop's value to *N*.

$$(1-1),(1-2),(1-3)\ \ldots\ (1-N),(2,1),(2,2)\ \ldots\ (N,N-1),(N,N)$$

The values above indicate the start and end point of each subgroup generated by the algorithm in the form:

$$(start\ point - \ end\ point)$$

Thereby, all connected subgroups are found.

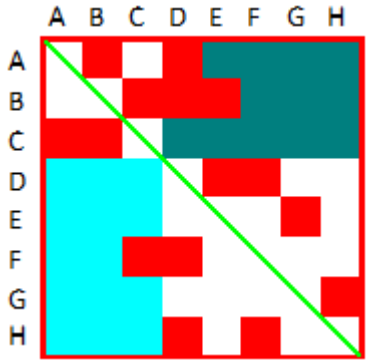


Figure 6-3 A DSM which has been cluster analysed.

A cluster analysed DSM displaying two distinct clusters can be seen in Figure 6-3. Removing the dependency in the light blue square, from F to C, will split the cyclic group.

6.4 Generating To-Be Scenarios

When the dependencies which are most valuable to remove have been highlighted a to-be scenario can be generated. The to-be scenario will display the resulting architecture if a chosen set of dependencies are removed. This can be used to evaluate the profit of breaking dependencies in relation to the cost of implementing such changes.

This is accomplished by removing the chosen dependencies in the original DSM. Then the VSM is recomputed with the resulting DSM.

6.5 Displaying Classification Changes of Components

A purpose of the to-be scenarios are to change the classification of components. If direct dependencies are removed from the Core the VFI_C and VFO_C values can change. As a result, the cyclic group sorting of the matrix changes. In this way components that

originally reside in Shared, Periphery or Control might get new classifications due to the new cyclic group sorting. The change in classification among components is illustrated by letting all components in the to-be scenario cyclic group sorted DSM keep their colours from the original cyclic group sorted DSM.

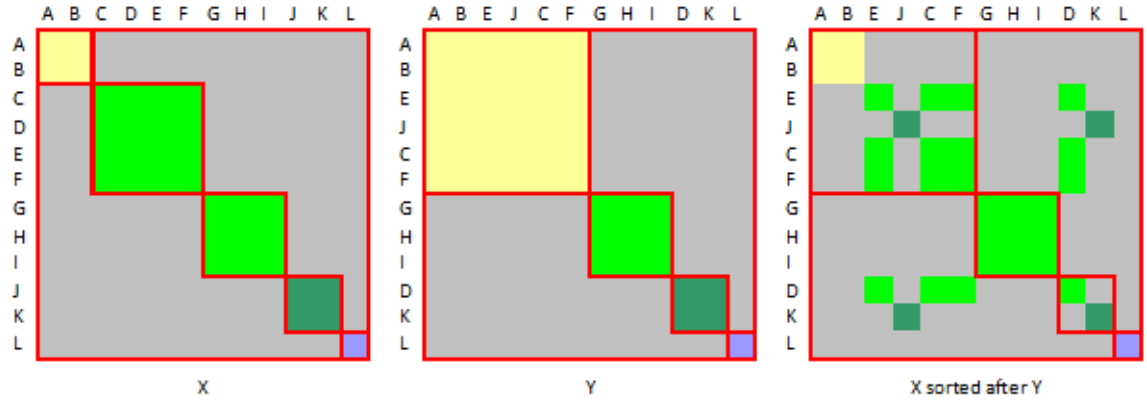


Figure 6-4 Illustration of what would happen if a cyclic group sorted DSM 'X' would be sorted after its to-be scenario cyclic group sorted DSM 'Y'. The red lines delineate the classification groups.

Components can in some cases appear to keep their classification. At first glance, when viewing DSM Y in Figure 6-4 it appears as if both components from the periphery group in DSM X have kept their classification. While in actuality, one periphery component 'J' has moved to the shared group.

7 Result

The result was generated by running the parser through the complete source code. The DSM, VSM, analyse matrix and theoretical to-be scenario were created.

The DSM that was generated with the data from the parser gave an architectural structure that was classified as Core-Periphery.

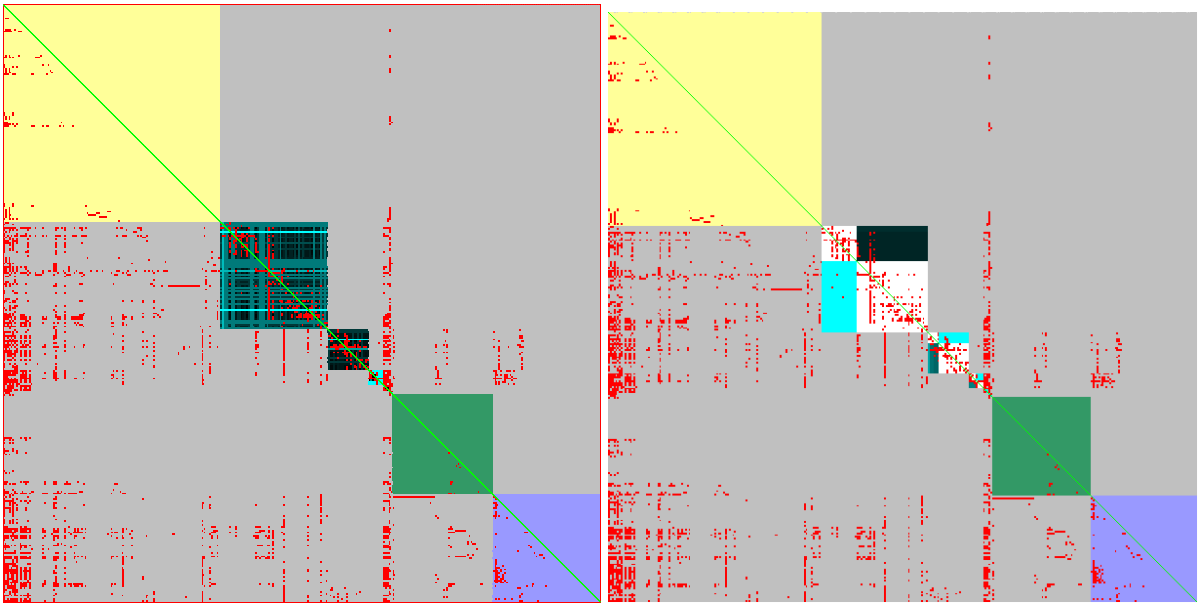


Figure 7-1 Two DSM:s displaying the architecture of the Ericsson software. The DSM on the left hand side displays the dominator analysis. The DSM on the right hand side displays the cluster analysis. Red dots are direct dependencies.

In Figure 7-1 , different classifications are displayed by the squares. The beige square in the top left corner represents the Shared components. The black squares with light blue lines represent the cyclic groups. The uppermost of the cyclic groups is the Core. The light blue lines indicate the FI and FO gain values in the different rows and columns. The green square represent periphery components. The blue square in the bottom right of the matrix represent the control components.

7.1 First To-Be Scenario

A to be-scenario was calculated by removing three edges. The removed dependencies were between components in the Core in Figure 7-1. They were located in the bright blue square with cluster analysis.

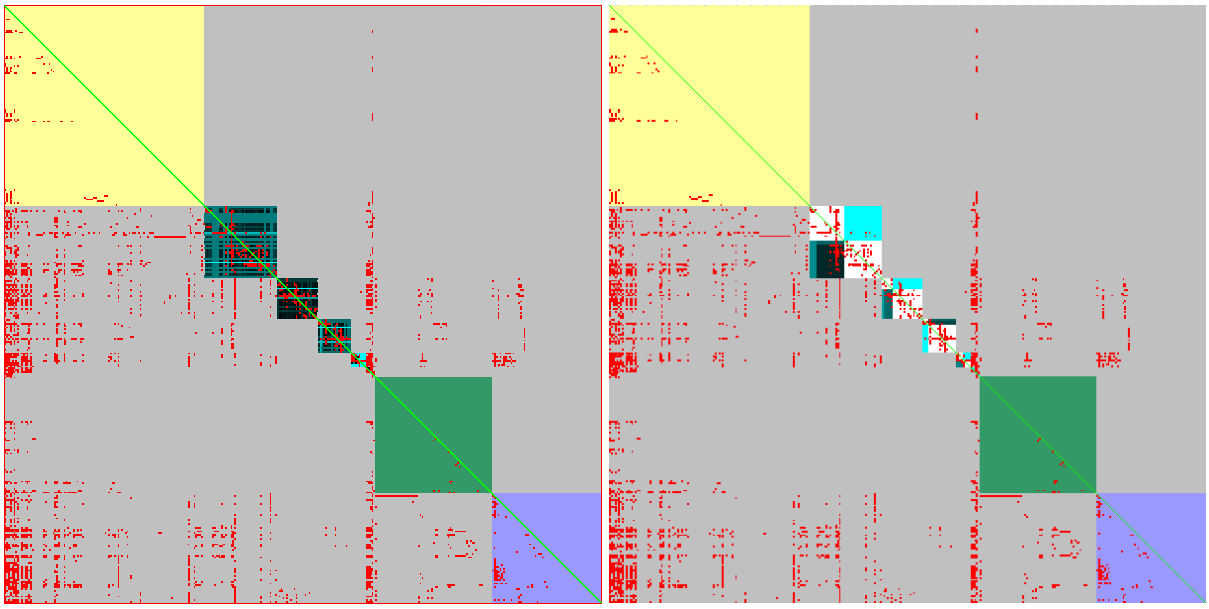


Figure 7-2 The resulting DSM for the first to-be scenario with a Core-Periphery classification. The DSM on the left hand side displays the dominator analysis. The DSM on the right hand side displays the cluster analysis.

7.2 Second To-Be Scenario

A to-be scenario was generated by removing five direct dependencies using cluster analysis. The dependencies were located in the bright blue square in the Core of the DSM from the first to-be scenario, see Figure 7-2.

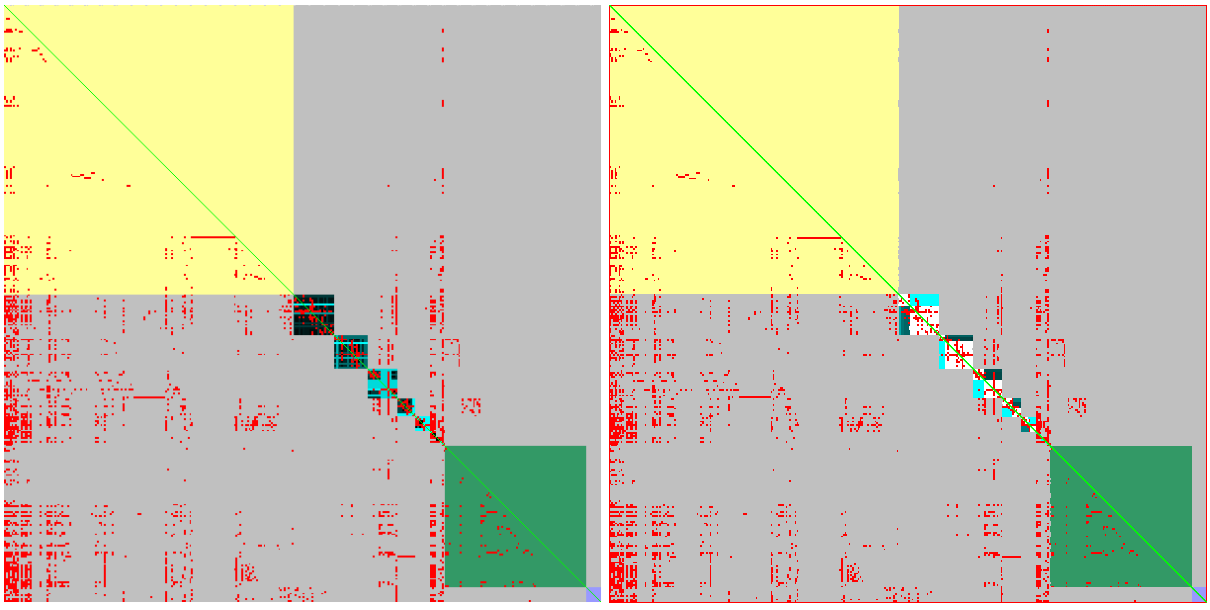


Figure 7-3 The resulting DSM for the second to-be scenario which obtained Multi-Core structure. The DSM on the left hand side displays the dominator analysis. The DSM on the right hand side displays the cluster analysis.

7.3 Third To-Be Scenario

Two dependencies were removed from the DSM in Figure 7-3 . One dependency from the Core and one from the second largest cyclic group. These were highlighted by the dominator analysis.

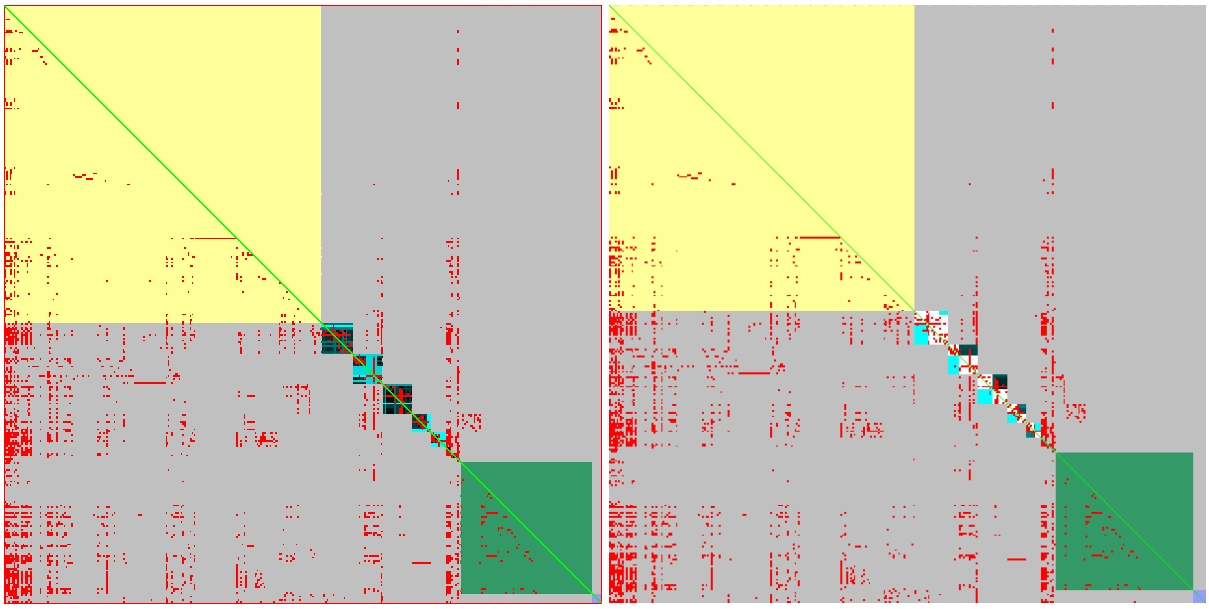


Figure 7-4 The resulting DSM for the third to-be scenario with Multi-Core architecture. The DSM on the left hand side displays the dominator analysis. The DSM on the right hand side displays the cluster analysis.

7.4 Fourth To-Be Scenario

Nine dependencies were removed from the DSM in Figure 7-4 . Highlighting was done using cluster analysis. Six of the dependencies resided in the Core, one in the second largest cyclic group and two in the third largest cyclic group.

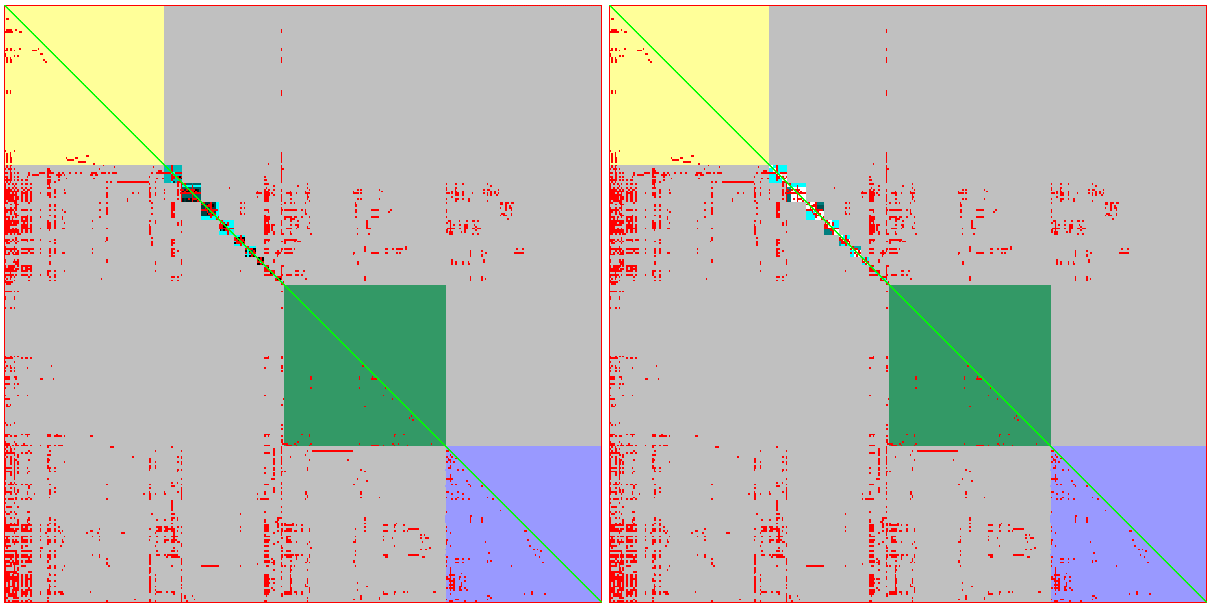


Figure 7-5 DSM displaying the fourth and final to-be scenario which obtained a hierarchical structure. The DSM on the left hand side displays the dominator analysis. The DSM on the right hand side displays the cluster analysis.

7.5 Table of Metrics from the To-Be Scenarios

Table 7-1 Metrics from the to-be scenarios. Self-dependencies are excluded from the direct dependencies since they cannot be altered. The percentages within parenthesis are relative to the original value.

Scenario	Figure 7-1	Figure 7-2	Figure 7-3	Figure 7-4	Figure 7-5
Shared	36,34%	33,54% (-7,70%)	48,45% (+33,32%)	51,24% (+41,00%)	26,71% (-26,49)
Core	18,01%	12,11% (-32,75%)	6,83% (-62,07%)	5,59% (-68,96%)	3,11% (-82,73%)
Second largest cyclic group	6,83%	6,83% (±0%)	5,59% (-18,15%)	4,97% (-27,23%)	3,11% (-54,46%)
Periphery	16,77%	19,57% (+16,69%)	23,60% (+40,72%)	22,98% (+37,03%)	27,02% (+61,12%)
Control	18,01%	18,32% (+1,72%)	2,48% (-86,22%)	2,17% (-87,95%)	26,09% (+44,86%)
Propagation cost	29,48%	26,25% (-10,95%)	22,31% (-24,32%)	21,67% (-26,49%)	19,75% (-33,00%)
Cluster cost	101656	84737 (-16,64%)	66470 (-34,61%)	62366 (-38,64%)	53723 (-47,15%)
Max depth	11	11	11	10	7
No. Direct Dependencies	2965	2962 (-0,10%)	2957 (-0,26%)	2955 (-0,33%)	2946 (-0,64%)
Architecture	Core-Periphery	Core-Periphery	Multi-Core	Multi-Core	Hierarchical
Average propagation cost reduction per direct dependency removed	-	3,65%	3,04%	2,65%	1,74%

8 Conclusions, Validity and Future Work

The parser gave a deeper understanding of the underlying architectural structure in Ericsson's source code. It is worth to note that all imports are counted as dependencies by the parser, even unused imports. This could be altered to only accept data accesses or method invocations in other components as dependencies. Such a parser might give a better view of which dependencies truly exist in the systems implementation.

Cluster analysis seemed to be advantageous to dominator analysis in most cases. It is highly dependent on how intertwined the components in the cyclic group are. A cyclic group which is closely coupled may be hard to split. In such cases dominator analysis works better.

Through applying cluster analysis, a Hierarchical structure was achieved by removing 22 dependencies which is seen in Appendix B. Dominator analysis obtained a Hierarchical structure by removing 26 dependencies which is seen in Appendix C.

. However, only 19 dependencies had to be removed to achieve similar results when both methods were combined. This indicates that both methods complement each other. The average propagation cost reduction per dependency removed decreased with each iteration. Conclusively, these techniques are most beneficial in the beginning since their effect diminishes as more dependencies are removed.

In the cyclic group analysis, both dominators and cluster analysis, are not dependent on the parser remaining in the same state. They can be used on any DSM. Hence they are viable techniques even if the parser is changed or replaced.

A drawback of the analysis is that dependencies in cyclic group might be hard to break. This can be an effect of early decisions regarding the structural design. These decisions might be hard to abolish if the targeted component is designed to be co-dependent.

A positive effect of having two analysis methods is that they in most cases recommended removal of different dependencies. This can be beneficial because it gives more alternatives on how the system can be improved.

8.1 The First To-Be Scenario

A view of the benefits of the cluster analysis is displayed in the first to-be scenario in Figure 7-1. All metrics from the to-be scenarios are displayed in **Error! Reference source not found.**. Three dependencies with the highest priority in the Core were removed. Even though only 0,10% of all direct dependencies were removed, the resulting architecture was greatly improved:

The propagation cost decreased with 10,95%, which will lower the chances for propagation of modifications and errors. This makes the system less complex and easier to comprehend since hidden dependencies are hard to detect.

The Core size was reduced by 32,75%. A large Core is an impeding feature and by reducing its size the system become more modular. All components in the Core are highly affected by change in the rest of the system. Hence, a smaller Core will make the development process more effective, especially for components in the Core.

8.2 The Multi-Core Structured To-Be Scenario

To obtain a Multi-Core structure, 0,26% of the direct dependencies in the original DSM had to be removed as can be seen in Figure 7-3. This structure is preferable over

Core-Periphery since it creates a more modular system. The two largest cyclic groups still contain a large part of the system, 12,11% and 6,83%. This is an impeding feature that requires coordination between development of components in the two cyclic groups.

The reduction in propagation cost indicates a great decrease of the probability for propagation of changes and hence a more modular system. This will reduce the need for updating pre-existing components due to propagation of modifications.

The relation between the amount of direct- and hidden dependencies which were removed shows the value of removing these dependencies: 0,26% of the direct dependencies generate 24,32% of all hidden dependencies.

As mentioned earlier, these dependencies may be closely intertwined in the system design. Therefore, they may be impossible to remove without a great reconstruction of the system.

8.3 The Hierarchical Structured To-Be Scenario

To achieve a Hierarchical structure 0,64% of all direct dependencies, excluding self-dependencies, had to be removed. This structure is preferable since it reduces the risk of generating errors in the system due to the architectural structure. The resulting DSM is displayed in Figure 7-5.

The propagation cost was reduced by 33%. It is a significant improvement in relation to the number of removed dependencies.

The size of the Core and the second largest group was reduced with 82,73% and 54,46% respectively. This would improve the possibility to distribute the development effort geographically. The probability of interference between development teams would also be reduced significantly.

By reducing the size of the cyclic groups to less than 4% of the system size, the system gets a clear hierarchy. Shared components have the highest position in the hierarchy. They are less vulnerable to being affected by changes in other components. Control components are highly affected by changes in the system and thus get a low position in the hierarchy. Thereby the planning process for the development becomes more manageable.

8.4 Validity of the Analysis

The key points of improvement in the original DSM show one dependency of special interest. This is a dependency previously known by development teams at Ericsson to be impeding. There have been discussions in order to get a reconstruction of the included components started.

This was not known to the project group when the analyse DSM was generated. Hence this helps to validate the methods for analysis.

8.5 Met Goals

- A parser used to extract dependencies from Ericsson's source code was implemented.
- A VSM, containing the hidden dependencies and hidden structures of the software architecture, was produced.
- Two methods were implemented to analyse and highlight key points for structural improvement in the VSM and DSM.
- To-be scenarios were generated using the two analyse methods developed in the thesis.
- The cost/gain ratio was estimated by the relation between removed direct dependencies and propagation cost.

8.6 Future Work

The cluster cost was calculated for the original scenario and all the to-be scenarios. However, the results could not be evaluated due to the lack of theoretical background on how to draw conclusions from cluster cost. Further studies might be conducted to develop the theory behind cluster cost.

Sorting the previous iterations of the cyclic group sorted DSM after newer iterations gave an obscure result when the DSM size was large. Classification groups of the DSM were in some cases difficult to delineate. The method aims to examine the change of classification for individual components. Thereby, to understand the result, the visualisation of the changes of classification should be improved.

The sorting algorithm for the cluster analysis is subject to change in future work. The algorithm gave a result that was applicable to the project, but the value of the result is hard to prove. This is due to the fact that "c*lustering is in the eye of the beholder*". A cluster analysis algorithm has to be evaluated in correlation to the problem at hand (Estivill-Castro, 2002). Development of a more fitting sorting algorithm, which separates clusters more clearly, would improve the analysis. The algorithm which highlights possible distinct subgroups is not dependent on the sorting algorithm. It can therefore still be used if the sorting is altered.

The algorithms could be further improved to highlight dependencies defying the design rules regarding a system structure. These dependencies should not exist and are valuable to examine in order to improve the system architecture.

The parser can be modified to only take active dependencies into account. Unused imports should not be viewed as dependencies in order to construct a correct model of a complex system.

The financial cost of the proposed to-be scenario, which achieved hierarchical structure, should be evaluated. The plausibility for removal of the dependencies, which are required to achieve the to-be scenario, could be interesting for Ericsson to examine.

Both the cluster- and dominators analysis methods should be validated by applying them to more systems in future empirical studies. The methods could also be tested on systems in other areas. An algorithm could be developed that evaluates which of the two methods is more beneficial in a specific scenario.

9 Acknowledgements

The thesis group received support from the supervisors at Ericsson:

- Mattin Addibpour
- Franz Heiser
- Svante Arvedahl
- Magnus Westerberg

The thesis group received support and inspiration from the examiner at KTH:

- Christian Schulte

The work was conducted collaborative by Oscar Bjuhr and Klas Segeljakt. All theories were discussed and implementation was done through pair programming. The work can roughly be split as follows:

Parser – Klas and Oscar

DSM and VSM processing in Excel – Klas

Solution to entangled cyclic groups - Oscar

Dominator analysis – Klas and Oscar

Cluster analysis - Oscar

Implementation of analysis in Excel – Klas and Oscar

To-be scenarios – Klas

10 References

Alan MacCormack, John Rusnak, Carliss Baldwin, 2006. *Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code,* Boston: Harvard Business School.

Appel, A. W., 2002. *Modern compiler implementation in Java*. Cambridge: Cambridge University Press.

Brian S. Everitt, Sabine Landau, Morven Leese, Daniel Stahl, 2010. *Cluster Analysis (5th ed., Wiley Series in Probability and Statistics)*. s.l.:Chicester : Wiley.

Carliss Baldwin, Alan MacCormack, John Rusnak, 2013. *Hidden Structure: Using Network Methods to Map Product Architecture,* Boston: Harvard Business School.

Edith Toma, Aybüke Aurum, Richard Vidgen, 2012. An exploration of technical debt. *Journal of Systems and Software,* p. 1498–1516.

Eppinger, S. D. & Browning, T. R., 2012. *Design structure matrix methods and applications*. s.l.:MIT Press.

Ericsson, 2016. *Company Facts*. [Online] Available at: http://www.ericsson.com/thecompany/company_facts

Estivill-Castro, V., 2002. Why so many clustering algorithms: a position paper. *ACM SIGKDD Explorations Newsletter,* 4(1), pp. 65-75.

Fenton, N. & Melton, A., 1990. Deriving structurally based software measures. *The Journal of Systems and Software,* 12(3), p. 177.

Floyd, R. W., 1962. Alogirthm 97: Shortest path. *Communications of the ACM,* 5(6), p. 345.

Giovanni, D., 1982. Technological paradigms and technological trajectories. *Research Policy,* Volume 11, pp. 147-162.

IEEE, 2016. *IEEE Policies,* New York: IEEE.

Information Sciences Institute, 1981. *Transmission Control Protocol - Darpa Internet Program Protocol Specification,* Southern California: s.n.

Johann Peter Murmann, Michael L. Tushman, 1997. Dominant designs, Technologicly cycles and Organizational outcomes. *Research in Organizational Behavior*.

Lozano, R., 2008. Envisioning sustainability three-dimensionally. *Journal of Cleaner Production,* 16(17), pp. 1838-1846.

Matthew J. LaMantia, Yuanfang Cai, Alan D. MacCormack, John Rusnak, 2008. *Analyzing the Evolution of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases*. s.l., IEEE, pp. 83-92.

Michael L. Tushman, Lori Rosenkopf, 1992. Organizational Determinants of Technological Change: Toward a Sociology of Technological Evolution. *Research in Organizational Behavior,* pp. 311-347.

Miguel A. Fortuna, Juan A. Bonachela, Simon A. Levin, 2011. Evolution of a modular software network. *Proceedings of the National Academy of Sciences,* pp. 19985-19989.

Munro, I., 1971. Efficient Determination of the Transitive Closure of a Directed Graph. *Information Processing Letters,* pp. 56-58.

Norman Fenton, Austin Melton, 1990. Deriving structurally based software measures. *Journal of Systems and Software,* pp. 177-187.

Parnas, D. L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* pp. 1053-1058.

Ray, S. S., 2012. *Graph theory with Algorithms and its applications*. s.l.:Springer India.

Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Julien Delange, Marco González, Philippe Kruchten, 2013. Variations on Using Propagation Cost to Measure Architecture Modifiability Properties. *IEEE International Conference on Software Maintenance,* pp. 400-403.

Robert Lagerström, Carliss Baldwin, Alan MacCormack, Stephan Aier, 2014. *Visualizing and Measuring Enterprise Application Architecture: An Exploratory Telecom Case*. s.l., IEEE, pp. 3847-3856.

Robert Lagerström, Mattin Addibpour, Franz Heiser, 2016. *Product Feature Prioritization Using the Hidden Structure Method: A Practical Case at Ericsson,* Stockholm: s.n.

Schilling, M. A., 2000. Toward a General Modular Systems Theory and Its Application to Interfirm Product Modularity. *The Academy of Management Review,* pp. 312-334.

Schneidewind, N. F., 2012. Operating Systems. In: *Computer, Network, Software, and Hardware Engineering with Applications, Software Engineering*. s.l.:Wiley-IEEE Press, pp. 286-302.

Sharman, D. M. & Yassine, A. A., 2004. Characterizing complex product architectures. *Systems Engineering,* 7(1), pp. 35-60.

Simon, H. A., 1962. The Architecture Of Complexity. *Proceedings of the American Philosophical Society,* 12 12.pp. 467-482.

Sosa, M. E., Eppinger, S. D. & Rowles, C. M., 2007. A Network Approach to Define Modularity of Components in Complex Products. *Journal of Mechanical Design,* 129(11), pp. 1118-1129.

Steward, D. V., 1981. The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on engineering management,* pp. 71-73.

Steven D. Eppinger, Daniel E. Whitney, Robert P. Smith, David A. Gebala, 1994. A model-based method for organizing tasks in product development. *Research in Engineering Design,* pp. 1-13.

Ulrich, K., 1995. The Role of Product Arhitecture in the Manufacturing Firm. *Research Policy,* 24(3), pp. 419-440.

Yu Zhifeng, V. Rajlich, 2001. *Hidden dependencies in program comprehension and change propagation*. Toronto, IEEE, pp. 293-299.

11 Appendix A

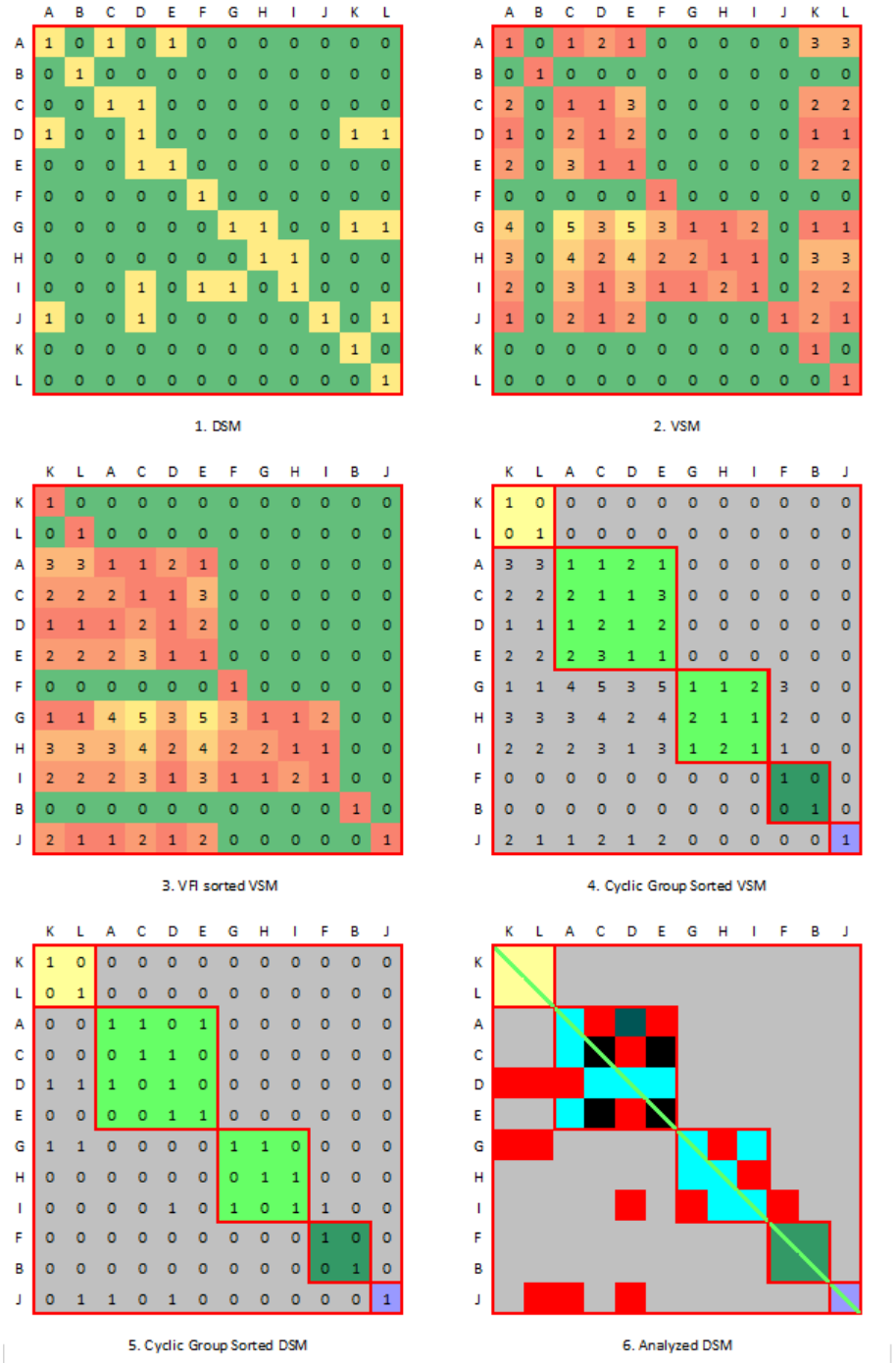


Figure 11-1 Entire process of generating a Dominator analysed DSM.

12 Appendix B

Table 12-1 Obtaining Hierarchical structure by only using cluster analysis.

Scenario	Original	1	2	3
Shared	36,34%	33,54%	29,50%	26,71%
Core	18,01%	12,11%	4,97%	3,11%
Second largest cyclic group	6,83%	5,59%	4,66%	3,11%
Periphery	16,77%	19,57%	23,60%	26,40%
Control	18,01%	20,81%	25,16%	27,33%
Propagation cost	29,48%	25,38%	20,78%	19,76%
Cluster cost	101656	79188	57090	53779
Max depth	11	10	7	7
No. Direct Dependencies	2965	2957	2948	2943
Architecture	Core-Periphery	Core-Periphery	Multi-Core	Hierarchical

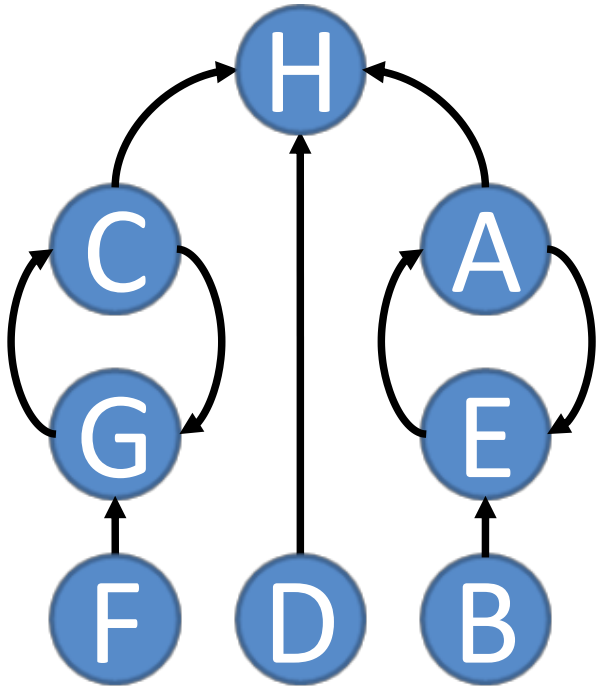
13 Appendix C

Table 13-1 Obtaining Hierarchical structure by only using dominators analysis.

To-Be scenarios	Original	1	2	3	4
Shared	36,34%	37,58%	36,96%	34,78%	52,48%
Core	18,01%	16,77%	16,15%	13,35%	3,73%
Second largest cyclic group	6,83%	5,59%	3,73%	3,73%	3,11%
Periphery	16,77%	16,77%	18,01%	20,19%	31,06%
Control	18,01%	19,25%	21,12%	23,91%	1,55%
Propagation cost	29,48%	28,50%	27,05%	24,96%	17,66%
Cluster cost	101656	98452	90004	79596	48412
Max depth	11	10	9	9	9
No. Direct Dependencies	2965	2962	2955	2951	2939
Architecture	Core-Periphery	Core-Periphery	Core-Periphery	Core-Periphery	Hierarchical

14 Appendix D

Two cyclic groups with the same VFI and VFO might become entangled when the VSM in generated. This will result in a much more unclear VSM.



	A	B	C	D	E	F	G	H
A	1	0	0	0	1	0	0	1
B	0	1	0	0	1	0	0	0
C	0	0	1	0	0	0	1	1
D	0	0	0	1	0	0	0	1
E	1	0	0	0	1	0	0	0
F	0	0	0	0	0	1	1	0
G	0	0	1	0	0	0	1	0
H	0	0	0	0	0	0	0	1

Figure 14-1 System with two entangled cyclic groups.

The solution for this is to sort all components after their highest placed dependency in the VSM. Height is measured as the highest position of a non-zero cell in a components column, the lowest point is the bottom of the picture and the highest point is the top.

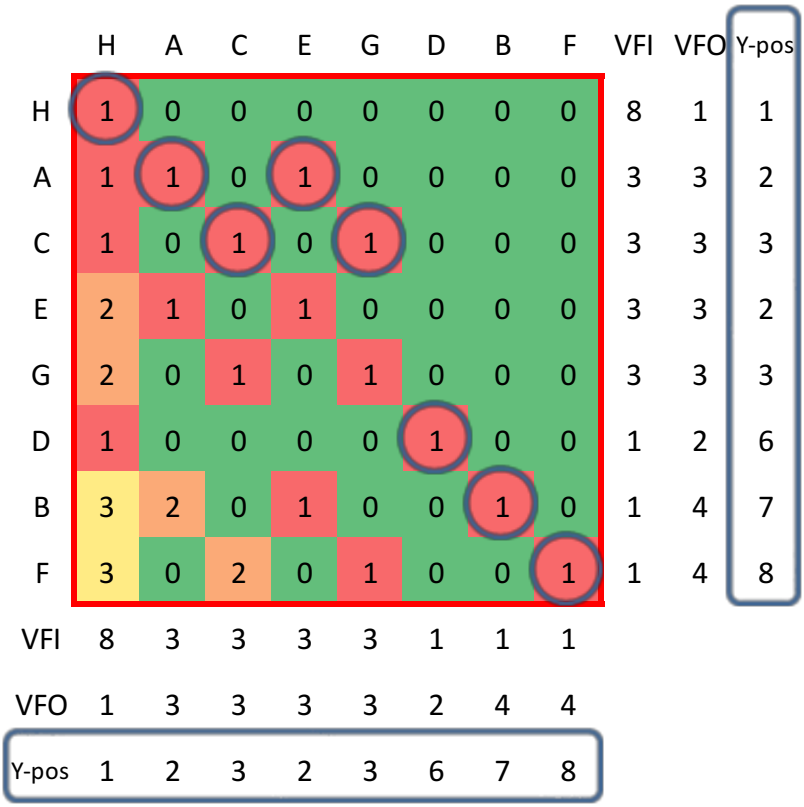


Figure 14-2 Picture illustrating the calculation of height.

All components outside the cyclic groups will have all dependencies placed beneath the diagonal. Hence, their highest located dependency in the VSM will be their self-dependence. Thereby, the order of the components outside the cyclic groups will be maintained. In a cyclic group, all components are indirectly dependent on each other. Thus the highest placed none zero cell for all components in the cyclic group will be the highest placed component in the group. Therefore, all components that are contained in the same cyclic group will be sorted after the same value and placed together. This will clearly delineate the entangled cyclic group without scrambling the rest of the components.

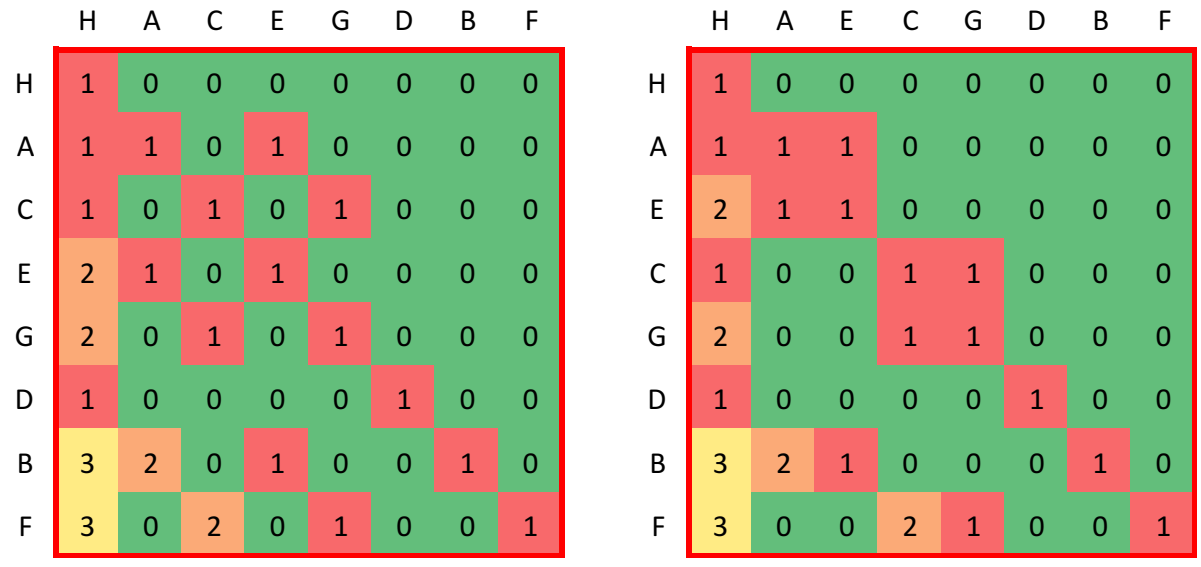


Figure 14-3 VFI sorted VSM to the left. VFI and height sorted VSM to the right.

TRITA-ICT-EX-2016:98