Modular software architecture for a large complex codebase

Citation for published version (APA):

Document status and date:
Published: 25/09/2015

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 16. Oct. 2019
Modular Software Architecture for a Large Complex Codebase

Lindung Manik
September 2015
Modular Software Architecture for a Large Complex Codebase

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

The design described in this report has been carried out in accordance with the TU/e Code of Scientific Conduct
Abstract
A large and complex codebase is evolving. Maintenance and change management become big problems. A more modular architecture is needed to improve maintainability. One of the improvement areas of the existing codebase is that, although it is divided into several building blocks, these building blocks are too intertwined. This report describes the approach of identifying the problems. It also gives the visualization to show how the code is intertwined. Besides providing problem analysis, the report also explains the solution design and the implementation to solve the problems. Moreover, future works are also addressed as recommendations, such as a way of working to prevent the problems from happening again and guidelines to understand the building blocks itself.

Keywords
modularity, architecture, dependency, restructuring, refactoring

Preferred reference
Preface

This report was created as a concluding deliverable of the project “Modular Software Architecture for a Large Complex Codebase”. This document provides a technical report of the graduation project of Software Technology PDEng program at Stan Ackermans Institute. The project was conducted over a period of nine months from January until September 2015.

The target audience of this report is a technical audience with a basic notation of modern software design and interested in modularity aspect. Readers that are interested in the global overview of the project can read the executive summary. Readers interested in the context and the problems should read Chapter 1 and Chapter 2. Chapter 3 gives an overview of reference literatures. Readers mainly interested in the proposed solution and the results are referred to Chapter 4 and Chapter 5. The future works and conclusions can be found in Chapter 6 and Chapter 7. Readers interested in the complete project are invited to read the entire report.

Lindung Manik
September 2015
Executive Summary

A large and complex codebase is evolving. Maintenance and change management become big problems. A more modular architecture is needed to improve maintainability. A nice modular structure makes it easier to understand and maintain the codebase. Failing to keep the structure clean can cause various maintainability problems, such as spaghetti code. Changes in one part of the codebase can break seemingly unrelated parts. This report describes a project to tackle the modularity problems.

One of the improvement areas of the existing codebase is that, although it is divided into several building blocks, these building blocks are too intertwined. To find the actual problems of the codebase, static analysis is performed by analyzing the dependencies between these building blocks. From the analysis result, it is concluded that the building block dependencies between the actual implementation in the codebase and the reference architecture are very different. The main problem is there are many unwanted dependencies.

The solution that is described in this report is designed to solve the dependency problems. It is proved that the solution is feasible to implement. An improved building block structure of the codebase is generated by using dependency structure matrix, restructuring strategy, and refactoring strategy. As the result, there is no more cyclic dependency.

Integrating static analysis in daily development can prevent the problems from happening again. One of the recommendations is to use the live dependency checker tools. It can trigger the awareness to developers if they start introducing bad dependencies. Keeping the structure of the building blocks clean could make the codebase easier to understand and to maintain. Another future work that also could be done after having nice structure of the building blocks is documenting the building blocks itself. The building block structure gives the developers a high level understanding of the codebase. However, a building block documentation that contains use-case diagram, package diagram, component diagram, sequence diagram, and class diagram would be very useful to give the developers detail understanding of the codebase.
# Table of Contents

Preface .................................................................................................................. v

Executive Summary ............................................................................................. vii

Table of Contents ............................................................................................... ix

List of Figures ....................................................................................................... xi

List of Tables ......................................................................................................... xiii

1. **Introduction** ................................................................................................. 1
   1.1 **Context** ..................................................................................................... 1
   1.2 **Outline** ..................................................................................................... 1

2. **Problem Analysis** ......................................................................................... 3
   2.1 **Codebase Analysis** .................................................................................. 3
   2.2 **Design Opportunities** .............................................................................. 4

3. **Literature Review** ......................................................................................... 7
   3.1 **Design Principle** ...................................................................................... 7
       3.1.1. The Dependency Inversion Principle (DIP) ........................................ 7
       3.1.2. The Acyclic Dependency Principle (ADP) ........................................... 7
   3.2 **Dependency Structure Matrix** .................................................................. 7
       3.2.1. Definition .............................................................................................. 7
       3.2.2. Algorithm ............................................................................................ 9

4. **Solution Design** .......................................................................................... 11
   4.1 **Dependency Problems Identification** ...................................................... 11
       4.1.1. Create Initial DSM ............................................................................. 11
       4.1.2. Transform DSM .................................................................................. 12
       4.1.3. Establish and Enforce Design Rules .................................................... 12
   4.2 **Restructuring Strategy** ............................................................................. 14
       4.2.1. Top-Down Approach ......................................................................... 14
       4.2.2. Bottom-Up Approach ....................................................................... 15
   4.3 **Refactoring Strategy** ................................................................................ 17
       4.3.1. Creating Interfaces ............................................................................. 17
       4.3.2. Breaking up Classes .......................................................................... 18
       4.3.3. Using Object Oriented Patterns ......................................................... 19
       4.3.4. Refactoring Result ............................................................................ 21

5. **Feasibility Analysis** ...................................................................................... 23
   5.1 **Impacts** .................................................................................................... 23
   5.2 **Risks** ....................................................................................................... 24
   5.3 **Results** .................................................................................................... 24
6. Future Work .................................................................25
  6.1 Integrating Dependency Checker Tool..............................25
  6.2 Understanding a Building Block ....................................27

7. Conclusions .................................................................29
  7.1 Results ............................................................................29
  7.2 Deliverables .....................................................................29
  7.3 Open Issues .....................................................................30
  7.4 Design Opportunities Revisited ......................................30

Bibliography .................................................................31
  References ............................................................................31
  Additional Reading ............................................................31

About the Authors ........................................................33
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Building Block Reference Architecture</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Building Block Dependencies Inside The Codebase</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Simple DSM</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The Dependency Graph of the MVC Architecture</td>
<td>8</td>
</tr>
<tr>
<td>Figure 5</td>
<td>MVC Architecture DSM</td>
<td>8</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Partitioned Simple DSM</td>
<td>9</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Lower Triangular DSM</td>
<td>9</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Initial DSM</td>
<td>11</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Transformed DSM</td>
<td>12</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Established Design Rules</td>
<td>12</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Enforced Rules</td>
<td>13</td>
</tr>
<tr>
<td>Figure 12</td>
<td>The DSM after Modifying Several Rules</td>
<td>14</td>
</tr>
<tr>
<td>Figure 13</td>
<td>BuildingBlock_A and BuildingBlock_H Dependency Graph</td>
<td>14</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Restructuring Impact after Moving Class.cs File</td>
<td>15</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Intertwined Building Blocks</td>
<td>16</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Cohesive Clusters of Files</td>
<td>16</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Bottom-up Restructuring Result</td>
<td>17</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Package Diagram of an Example of Cyclic Dependency Problem</td>
<td>17</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Class Diagram of the Cause of Cyclic Dependency Problem</td>
<td>17</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Resolving Cyclic Dependency Problem by Creating Interface</td>
<td>18</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Resolving Cyclic Dependency Problem by Creating Interface</td>
<td>18</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Package Diagram of Another Example of Cyclic Dependency</td>
<td>18</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Class Diagram of Another Example of Cyclic Dependency</td>
<td>19</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Resolving Cyclic Dependency Problem by Breaking up Class</td>
<td>19</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Resolving Cyclic Dependency Problem by Breaking up Class</td>
<td>19</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Class Diagram of the Cause of Cyclic Dependency Problem</td>
<td>20</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Package Diagram of an Example of Cyclic Dependency Problem</td>
<td>20</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Resolving Cyclic Dependency by Using Factory Pattern</td>
<td>20</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Resolving Cyclic Dependency by Using Factory Pattern</td>
<td>21</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Files Distribution in the BuildingBlock_W</td>
<td>21</td>
</tr>
<tr>
<td>Figure 31</td>
<td>ClassX, ClassY, and ClassZ</td>
<td>23</td>
</tr>
<tr>
<td>Figure 32</td>
<td>Interface, ClassX, ClassY, and ClassZ</td>
<td>23</td>
</tr>
<tr>
<td>Figure 33</td>
<td>Alternative Approach of Extracting Abstraction</td>
<td>23</td>
</tr>
<tr>
<td>Figure 34</td>
<td>DSM after Restructuring and Refactoring</td>
<td>24</td>
</tr>
<tr>
<td>Figure 35</td>
<td>Class Diagram of the LiveNDepCheck</td>
<td>25</td>
</tr>
<tr>
<td>Figure 36</td>
<td>What Developers Want</td>
<td>27</td>
</tr>
</tbody>
</table>
List of Tables

Table 1 – The Color Changes of the Transformed DSM after Enforcing Rules… 13
Table 2 – Clustering Result ................................................................. 16
Table 3 – Example of NsDepCop Rules ............................................. 26
Table 4 – Example of LiveNDepCheck Rules ................................. 26
Table 5 – Example of LiveNDepCheck Ignored Rules ....................... 27
1. Introduction

Abstract – This chapter gives a general introduction to the context of the project and an overview of the structure of the report.

1.1 Context

A large and complex codebase is evolving. Maintenance and change management become big problems. A more modular architecture is needed to improve maintainability. A nice modular structure makes it easier to understand and maintain the codebase. Failing to keep the structure clean can cause various maintainability problems, such as spaghetti code. Changes in one part of the codebase can break seemingly unrelated parts.

The codebase that this project focuses on has two kinds of structures. They are organizational software structure and physical software archive structure. Organizationally, the codebase is divided into several function clusters. A function cluster could be divided into several building blocks. A building block could be divided into several components. A component contains one or more files. The scope of this project is only specific to a function cluster.

The inputs of this project are building blocks architecture of the function cluster as the reference architecture and the source code files for each building block. A building block is the key element in the software architecture. Building blocks are main units of design for the software development. The building block decomposition enables commonality and reuse. The Function Cluster Architect (FCA) has a central role with regard to the decomposition of a function cluster in terms of building blocks.

One of the improvement areas of the existing codebase is that, although it is divided into several building blocks, these building blocks are too intertwined. The goals of this project are:

- Current codebase analysis and visualization to show how the code is intertwined
- Guidelines to improve the software architecture modularity
- A compilable prototype according to the new proposed structure and guidelines

1.2 Outline

The report is structured as follows:

- Chapter 2 (Problem Analysis) explains the problem that this project is solving. It also describes design aspects that were selected as criteria for evaluating the results.
- Chapter 3 (Literature Review) explains the main inputs that are gathered by reading some literature on the project topic.
- Chapter 4 (Solution Design) explains the design of the solution by using the principles and methodology that is gained in the literature review to resolve dependency problems that are described in the problem analysis.
- Chapter 5 (Feasibility Analysis) describes the impacts and the risks of suggested improvement. It also describes the preliminary results after implementing the solution design.
- Chapter 6 (Future Work) gives a brief explanation of tools that can be used to prevent the dependency problems from happening again. It also describes guidelines to understand a building block after we have cleaned dependencies.
- Chapter 7 (Conclusion) summarizes the achievements of this project together with the main deliverables of the project. Moreover, design criteria that are mentioned in Chapter 2 are revisited to investigate if they were successfully addressed.
2. Problem Analysis

Abstract – This chapter describes the modularity problem that this project is solving. It shows the analysis result on the codebase and visualize how the code is intertwined. It also describes design aspects that were selected as criteria for evaluating the results.

2.1 Codebase Analysis

We use the static analysis method to identify the modularity problems inside the codebase of the function cluster. Static analysis is an analysis of software that is performed without actually executing programs. The goal of the analysis is to identify dependency problems between the building blocks. The function cluster codebase consists of 22 building blocks and 2792 files. To analyze this large codebase, help from a static analysis tool is needed.

However, the current physical software archive structure does not reflect the organizational structure. The software is written in C# .NET programming language. Many available static analysis tools in the market that support .NET, such as NDepend, Structure101, Lattix, or Resharper, can only analyze a compiled codebase based on its assemblies or its namespaces. Since an assembly or a namespace of the codebase does not represent a building block, assembly or namespace based analysis is not desirable. However, there is one tool that has the unique capability to perform the analysis based on the source code files, namely Understand.

Before performing the analysis, the FCA defined the building block architecture, which is shown by Figure 1. The FCA also grouped the source code files into the building blocks based on his knowledge and his expectation.

![Figure 1 – Building Block Reference Architecture](image-url)
Understand is used to analyze the codebase. It can export the actual implementation of the building block architecture dependencies from the codebase. The output is represented by Figure 2. Based on the output, it could be easily concluded that the implementation is very different from the architecture. There are several unwanted cyclic dependencies between building blocks that are represented by red lines although there is no cyclic dependency in the reference architecture. Moreover, even though all intended dependencies, which are represented by black lines, are present, many dependencies that exist in the implementation are not intended in the reference architecture. These unwanted dependencies are represented by blue lines.

This project focuses on solving these dependency problems. After this problem is resolved, other questions also arise, such as:

- How can we prevent the dependency problems from happening again?
- How can we understand a building block?

### 2.2 Design Opportunities

In this section, three design aspects that were selected to be used in evaluating the project design are presented. In addition, two design aspects were identified as not relevant to the project. The three design criteria that are relevant to the context of the project are:

- **Genericity** – Based on the experiences that are gained during the execution of the project, the findings will be applied not only to the current codebase but also to other large and complex codebases. Therefore, the design has to be generic enough to support all codebases.
- **Realizability** – The project serves as a proof of concept for a better codebase. Therefore, the goal is to prove whether it is feasible to realize the improvement proposal. Since the software is very complex, the improvement should not ruin the current codebase that is already working well. The pro-
ject aims to improve a complex codebase by implementing a noncomplex design. Since the complex steps tend to break the working codebase, the improvement has to be simple.

- **Documentation** – The project needs to provide documentations that describe all findings, such as the guidelines of certain process, the analysis, and also the results. Since this is a feasibility study project, the artifacts would be useful for the software development team in the future.

The two design criteria that are not relevant to the context of the project are:

- **Inventiveness** – Modularity was introduced in 1970s. Many metrics, tools, and methods have been proposed to resolve modularity problems since then. Thus, this project would not bring something new but it uses the existing metrics, tools, and methodology with little modifications.

- **Functionality** – No new feature is introduced in the software during the execution of this project. However, the improvement should not break the current functionalities.
3. Literature Review

Abstract – Having a clear definition of the problem requires clear inputs. This chapter presents a small part of the literature as the project references. The codebase architecture is a layered architecture. Thus, the first section of this chapter describes the design principles in modularity aspects of a layered architecture. The second section of this chapter describes a tool that is used to visualize the implementation of the layered architecture and the reason behind the choice of the tool. The design principles and the visualization tool are used to make the solution design in order to solve the dependency problem.

3.1 Design Principle

Martin (2003) writes several design principles and design patterns of software architecture in Agile Software Development: Principles, Patterns, and Practices book. These principles and patterns can be used to prevent software from starting to rot. Software tends to be large networks of interrelated packages. Changes that introduce new and unplanned package dependencies cause designs to rot. In this section, two dependency principles are explained as the main inputs of the project.

3.1.1. The Dependency Inversion Principle (DIP)

This principle states “depend upon abstractions, do not depend upon concretions.” Dependency inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. The implication of this principle is quite simple. Every dependency in the design should target an interface or an abstract class. No dependency should target a concrete class. As much as feasible, the principle should be followed. The reason is simple; concrete things change a lot but abstract things change much less frequently.

3.1.2. The Acyclic Dependency Principle (ADP)

One of rules that govern interrelationships between packages in software architecture is The Acyclic Dependency Principle (ADP). The principle states “dependencies between packages must not form cycles.” This implies that the dependencies must form a directed acyclic graph. It is in general always possible to break a cyclic dependency chain. The cycles can be broken in two ways. The first involves creating a new package, and the second makes use of the DIP.

3.2 Dependency Structure Matrix

3.2.1. Definition

Dependency Structure Matrix (DSM) is a simple tool to perform both the analysis and the management of a complex system. As a tool for system analysis, DSM provides a compact and clear representation of a complex system and a capture method for the interactions/interdependencies/interfaces between system elements (i.e., subsystems and modules). A DSM is a square matrix (i.e., it has an equal number of rows and columns) that shows relationships between elements in a system. One primary benefit of DSM is the graphical nature of the matrix display format. DSM has been widely used in the analysis of manufacturing processes. It was invented for optimizing product development processes, although it has broader applications – including project management and software. The term ‘dependency structure matrix’ refers both to a particular representation of such dependencies and to algorithms for reorganizing the dependencies by reordering and clustering elements. The matrix is a simple adjacency matrix with elements labeling the horizontal and vertical axes, and a mark in the $i^{th}$ row and $j^{th}$ column when the $i^{th}$ element depends on the $j^{th}$.
Dependencies of elements on themselves are not considered, so there are never marks along the diagonal. The strength of the dependency is given numerically. One important criterion that is used to evaluate the matrix is that the dependency relation should be acyclic based on the ADP, which is explained in Section 3.1.2. This means, in matrix terms, that the elements can be permuted so that the matrix is lower triangular – that is, with no entries above the diagonal.

Figure 3 shows a simple DSM in the development of a product. These tasks have dependencies on one another, either because of physical objects that must flow from task to task or because of information that one task requires and which another task provides. If the tasks are tightly coupled, with many cyclic dependencies, the pipeline stalls frequently, and tasks need to be repeated because of dependencies on tasks that follow them. We read the dependency across the row. Examining column 1, we note that task C depends on task A; examining column 3, we note that task A depends on task C. Because tasks A and C are mutually dependent, the tasks cannot be reordered to make the matrix lower triangular.

Figure 3 – Simple DSM

Sangal (2005) described the application of the DSM to software when he introduced Lattix. It has been primarily used in the field of software/IT systems engineering across many industries. The application of the DSM to software is pretty straightforward, with modules playing the role of tasks. In the context of this project, building blocks play the role of tasks.

Talking of dependencies of MVC software architecture means talking about something that looks like Figure 4. The picture shows the dependency graph of the architecture. In the form of the DSM, the dependency of the architecture should look like Figure 5.

Figure 4 – The Dependency Graph of the MVC Architecture

Figure 5 – MVC Architecture DSM
As a rule by default, an element can depend on other elements that lay on top of it but not the other way around:
1. Controller module can depend on View and Model modules.
2. View module can depend on Model module but View module cannot depend on Controller module.
3. Model module cannot depend on View and Controller modules.

In this report, the DSM is used more than the dependency graph. Although the DSM and the dependency graph are used to represent the same information, there is a trade-off:
- The dependency graph is more intuitive but can be totally not understandable when the numbers of nodes and edges grow. A dozen of building blocks can be enough to produce a graph too complex.
- The DSM is less intuitive but can be very efficient to represent large and complex graph.

3.2.2. Algorithm
There are several algorithms that can be used to reorganize the DSM in order to optimize the ordering of elements and their aggregation into groups. One of the algorithms is the partitioning algorithm. An example of the usage of this algorithm could be applied to our example in Figure 3. Figure 6 and Figure 7 show the DSM after we apply the algorithm to the Simple DSM. If tasks A and C are considered as a single composite task, the cycle can be eliminated.

![Partitioned Simple DSM](image)

![Lower Triangular DSM](image)

Another type of algorithm is the clustering algorithm. It optimizes the ordering and aggregation to reduce the number off-diagonal dependencies. Its purpose is not merely to eliminate cycles, but also to reduce the incidences of any dependencies between clusters. ■
4. Solution Design

Abstract – This chapter describes the design of the solution by using the principles and methodologies that are gained in the literature review to resolve dependency problems which are described in the problem analysis.

There are four steps as the general approach to resolve dependency problems in a large software system. They are:

1. Create initial DSM
2. Transform DSM
3. Establish and enforce design rules
4. Improve structure

The first step could be done automatically by using the static analysis tool while the rest of the steps should be done manually because they need thorough analysis. The first section of this chapter describes the first three steps. These steps are used to identify all dependency problems by using DSM visualization. The second section and the third section describe the last step to solve the problems. There are two strategies to improve the structure. They are restructuring and refactoring strategies.

4.1 Dependency Problems Identification

4.1.1. Create Initial DSM

The first step is creating the initial DSM. The goal of this step is to extract dependencies from the code in a form of DSM. Figure 2 already shows the dependencies. However, it is not readable. By using static analysis tool, such as Understand, we can obtain the DSM as another representation of the dependencies. The tool extracts the DSM in CSV format. Microsoft Excel is powerful enough to open the output file. However, additional efforts are required to put colors in the matrix cells in order to beautify the visualization. In this project, the cells were colored manually. However, it could be automated in the future by adding a few rules to the sheet. The red cells represent the entries that are involved in cyclic dependencies while blue cells represent the entries that are involved in acyclic dependencies. Figure 8 shows the DSM. By default, the elements in DSM are ordered alphabetically.

![Initial DSM](image)
4.1.2. Transform DSM
The second step is transforming the initial DSM. The goal of this step is to organize the matrix by moving its rows or its columns so that the DSM can reflect “should be” architecture. The elements are reordered based on the hierarchical structure of the intended layered architecture, which is shown by Figure 1. Higher level building block, which depends on the lower level building block, is placed at the bottom of DSM while lower level building block, which is used by the higher level building block, is placed at the top of DSM. Figure 9 shows the transformed DSM.

![Figure 9 - Transformed DSM](image)

4.1.3. Establish and Enforce Design Rules
The third step is establishing and enforcing the design rules. As the first step, we make a copy of the transformed DSM in Figure 9. After that, we remove all entries, including the colors and the numbers. As the second step, we establish the rules in the DSM. Figure 10 shows all rules.

![Figure 10 - Established Design Rules](image)
We put color in the cells below diagonal to represent the rules. A blue color represents an allowed dependency while a yellow color represents a disallowed dependency. We can obtain the rules from the reference architecture in Figure 1. Every arrow in Figure 1 is a blue cell. The BuildingBlock_K and the BuildingBlock_Q are special cases. According to the FCA, these building blocks can be used by others even though there is no arrow in the reference architecture. Then, we put yellow color in the rest cells other than blue cells. There should be no entries in the yellow cells.

After we have the established design rules, now we can enforce them to the transformed DSM to catch rule violations. The goal of this step is to change the color of each cell in the transformed DSM. We pair each cell below diagonal of the transformed DSM with the corresponding cell in the design rules. We do not need to change the color of the cells above diagonal because they are violations by default. Table 1 shows the color changes that are applied in the transformed DSM after we enforce the rules.

<table>
<thead>
<tr>
<th>The color of the cell in the transformed DSM</th>
<th>The color of the cell in the design rules</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>Blue</td>
<td>Blue</td>
</tr>
<tr>
<td>Blue</td>
<td>Yellow</td>
<td>Orange</td>
</tr>
<tr>
<td>Red</td>
<td>Blue</td>
<td>Red</td>
</tr>
<tr>
<td>Red</td>
<td>Yellow</td>
<td>Darker red</td>
</tr>
</tbody>
</table>

Figure 11 shows the results. Only blue cells below diagonal comply with the rules. As we can see from the picture, there are three kinds of rule violations:

- Violations of the ADP principle (the cyclic dependencies) that are shown by red cells.
- Violations of the default rules that are shown by entries above the diagonal.
- Violations of the established design rules that are shown by orange cells.

Darker red cells represent the dependencies that violate two rules at once, which are the violation of the ADP principle and the violation of the established design rules. There should be no entry in those cells but there is an entry there and these entries are involved in cyclic dependencies as well.

![Figure 11 - Enforced Rules](image-url)
In this step, we also can modify the rule if it is needed. For instance, orange cell could be turned into blue cell if the dependency is very strong. The strength of the dependency is shown by the number inside the cell. As an example, the BuildingBlock_R uses the BuildingBlock_E in 2827 occasions. We change the color of the cell that is located in row 18 and column 5 from orange to blue because it turns out that the BuildingBlock_R can depend on the BuildingBlock_E. An error in the reference architecture is found at this point. This means that the dependency should be added to Figure 1 as well. Figure 12 shows the DSM after modifying several rules.

![Figure 12 – The DSM after Modifying Several Rules](image)

After we identify all violations of the rules, the next step is eliminating these unwanted dependencies.

### 4.2 Restructuring Strategy

Restructuring strategy means reorganizing the codebase without modifying the code to get rid of unwanted dependencies. Restructuring is the strategy of moving the classes as much as possible rather than changing code. There are two approaches in the restructuring strategy. They are the top-down approach and the bottom-up approach.

#### 4.2.1. Top-Down Approach

This approach is used to resolve dependency problems based on reference architecture. For example, we want to eliminate the dependency between the BuildingBlock_A and the BuildingBlock_H because the dependency is not intended by the architect. Firstly, we have to look at the dependency graph between these two building blocks, as shown in Figure 13.

![Figure 13 – BuildingBlock_A and BuildingBlock_H Dependency Graph](image)
As we can see, the BuildingBlock_A uses the Class.cs file in the BuildingBlock_H. Secondly, we have to investigate the source code of the file. In this example, the file contains error code constants. It needs the expert’s judgment to decide whether the file is in the right building block or not. In this case, the file is not in the right place because error code constants are not specific to BuildingBlock_H functionality. It is a general code that can be used by other functionalities. The file in the wrong building block is the source of unwanted dependency.

Therefore, we should move the file to the BuildingBlock_K because this building block is used as a general infrastructure of the codebase. After we move the Class.cs file to BuildingBlock_K, we can see the result in Figure 14. This restructuring has huge positive impact, not only to the BuildingBlock_A but also to other building blocks. Several building blocks such as the BuildingBlock_M, the BuildingBlock_E, the BuildingBlock_N, and the BuildingBlock_P do not depend anymore on the BuildingBlock_H. The same file in different building blocks makes a very different dependency.

<table>
<thead>
<tr>
<th>Dependent Architecture</th>
<th>11</th>
<th>17</th>
<th>3</th>
<th>21</th>
<th>7</th>
<th>15</th>
<th>16</th>
<th>9</th>
<th>14</th>
<th>22</th>
<th>19</th>
<th>5</th>
<th>13</th>
<th>4</th>
<th>18</th>
<th>10</th>
<th>12</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildingBlock_K</td>
<td>11</td>
<td>17</td>
<td>3</td>
<td>21</td>
<td>7</td>
<td>15</td>
<td>16</td>
<td>9</td>
<td>14</td>
<td>22</td>
<td>19</td>
<td>5</td>
<td>13</td>
<td>4</td>
<td>18</td>
<td>10</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>BuildingBlock_Q</td>
<td>17</td>
<td>11</td>
<td>17</td>
<td>3</td>
<td>21</td>
<td>7</td>
<td>15</td>
<td>16</td>
<td>9</td>
<td>14</td>
<td>22</td>
<td>19</td>
<td>5</td>
<td>13</td>
<td>4</td>
<td>18</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>BuildingBlock_C</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>BuildingBlock_U</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>BuildingBlock_G</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>BuildingBlock_O</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>BuildingBlock_T</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>BuildingBlock_P</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>BuildingBlock_I</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>BuildingBlock_H</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>BuildingBlock_M</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>BuildingBlock_E</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BuildingBlock_N</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>BuildingBlock_P</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>BuildingBlock_F</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>BuildingBlock_M</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>BuildingBlock_D</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>BuildingBlock_R</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>BuildingBlock_J</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>BuildingBlock_L</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

**Figure 14 – Restructuring Impact after Moving Class.cs File**

### 4.2.2. Bottom-Up Approach

The bottom-up approach is used by ignoring the reference architecture. We restructure the building blocks by recursively grouping a cohesive cluster of files. The steps are:

1. Group all files in one large container
2. Find the cohesive cluster of files automatically
3. Wrap the cohesive cluster of files into small containers manually
4. Find the cohesive clusters of small containers automatically
5. Wrap cohesive clusters of small containers into higher level containers
6. Repeat steps 4 and 5 until we are satisfied with the result

In Figure 15, we can see the intertwined building blocks. There are more than 800 files inside these building blocks. The top-down approach cannot be used anymore to disentangle cyclic dependencies between these building blocks because moving files from one building block to another just makes new dependency problems. We need to make new building block architecture for these files with fewer cyclic dependencies. First, we group all files in one folder regardless where they belong to. After that, we analyze the dependencies between files by using static analysis tool to get the
DSM. We can apply a clustering algorithm to get cohesive clusters of files. There are several available tools that have a feature to generate the output automatically. In this example, the Lattix is used to generate the result, as shown by Figure 16.

![Figure 15 – Intertwined Building Blocks](image1)

![Figure 16 – Cohesive Clusters of Files](image2)

After we get the cohesive cluster of files, we can wrap them into small containers and then wrap the small containers into higher level containers. Table 2 shows the clustering result and Figure 17 shows the DSM result. There is no cyclic dependency anymore. However, there is a big cluster, which contains more than 500 files. We name it BuildingBlock_W. The other clusters are named from the reference architecture. As we can see, several building blocks such as BuildingBlock_Q and BuildingBlock_C are missing. The files of those building blocks including several files from other building blocks are clustered in the BuildingBlock_W.

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Number of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildingBlock_K</td>
<td>61</td>
</tr>
<tr>
<td>BuildingBlock_W</td>
<td>506</td>
</tr>
<tr>
<td>BuildingBlock_U</td>
<td>8</td>
</tr>
<tr>
<td>BuildingBlock_P</td>
<td>38</td>
</tr>
<tr>
<td>BuildingBlock_O</td>
<td>15</td>
</tr>
<tr>
<td>BuildingBlock_G</td>
<td>118</td>
</tr>
<tr>
<td>BuildingBlock_H</td>
<td>29</td>
</tr>
</tbody>
</table>
We need to break the BuildingBlock_W in smaller ones. It is big because the files inside of it are too intertwined. We need another strategy that is called as the refactoring strategy to disentangle this big building block.

4.3 Refactoring Strategy

Refactoring strategy means changing code without modifying behaviors, for example by creating interfaces, breaking up classes, and using object oriented patterns (Fowler, 1999). To choose which method to be used is depending on the context of the problem. Sometimes, we have to combine these methods in order to solve the problem. As the rule of thumb, we have to select the smallest efforts. The efforts can be calculated by counting the affected classes or lines of code if we use a certain method.

4.3.1. Creating Interfaces

This approach is the implementation of the DIP, which is explained in Section 3.1.1. As an example, we want to break cyclic dependency, which is represented by Figure 18 and Figure 19, between the BuildingBlock_Q and the BuildingBlock_P by creating interfaces.

Figure 18 – Package Diagram of an Example of Cyclic Dependency Problem

Figure 19 – Class Diagram of the Cause of Cyclic Dependency Problem
One of the solutions to get rid of this dependency is to create interfaces for Class1 as well as Class2 and put them in lower layer building block, for example, the BuildingBlock_K. The solutions are shown by Figure 20 and Figure 21.

However, restructuring the building blocks by moving the Class1 and the Class2 from the BuildingBlock_P to the BuildingBlock_Q also solves the dependency problem. This approach has lower effort than creating interfaces because we do not change anything in the code. We do not have to add new interfaces or modify existing classes. Thus, we opt to choosing the restructuring strategy in order to solve this problem.

### 4.3.2. Breaking up Classes

This approach is used to break up classes to separate responsibilities. For instance, we want to break cyclic dependency between the ClassA and the ClassB, which is represented by Figure 22 and Figure 23.
Figure 23 – Class Diagram of Another Example of Cyclic Dependency

One of the solutions to remove cyclic dependencies for this example is by moving the ClassA’s operations to a new class, and then putting the class in a new building block. Figure 24 shows the class diagram of the solution. We create a new class, namely ClassC. After that, we move two operations from the ClassA into the ClassC.

Figure 24 – Resolving Cyclic Dependency Problem by Breaking up Class

Figure 25 shows the new building block structure. We create a new building block, namely BuildingBlock_X, and we put the new class in the new building block.

Figure 25 – Resolving Cyclic Dependency Problem by Breaking up Class

4.3.3. Using Object Oriented Patterns

Besides the design principles, Martin (2003) also shows several design patterns that can be used to break cyclic dependencies. In the following example, we break cyclic dependency by using Factory pattern. The DIP strongly recommends that modules do not depend upon concrete classes. However, in order to create an instance of class, we must depend upon concrete classes. Factory pattern is a pattern that allows dependency upon the concrete class to exist in one, and only one, place.

As an example of a problem, we can look at the evolution of the codebase from Figure 24 to Figure 26. At some point, the developer introduced the ClassF in the BuildingBlock_U. Whether the developer realized or not, he or she introduced a cyclic dependency between the building block and the ClassC, which is represented by Figure 27. The ClassF uses the ClassC to reuse its functionality.
One of solutions to resolve the cyclic dependency is by moving out specific functionality from the ClassC that is used by the ClassF. As the first step, we create an interface for the ClassB, the ClassD, and the ClassE. We name it IClass. Judging from the operation name, these classes have a same behavior. The abstraction of these classes is extracted to the IClass.

The impact of the first step is the ClassF does not depend on the ClassC anymore. Both the ClassF and the ClassC can use the interface. However, these classes need an instantiation of the interface. Thus, as the second step, by using Factory pattern, we create a factory class, namely FactoryClass, to instantiate one of the concrete classes of IClass. As the last step, we put the IClass to the BuildingBlock_W and the FactoryClass to the BuildingBlock_U. The solution is shown by Figure 28 and Figure 29.
4.3.4. Refactoring Result

Figure 30 shows the files distribution of the BuildingBlock_W after the refactoring.

The number of files is decreased from 506 files to 127 files. More than 300 files are disentangled from the BuildingBlock_W building block. As much as possible, we put back those files into their original building block. 124 files are moved to the BuildingBlock_Q, 21 files are moved to the BuildingBlock_C, and the rest of files are moved to other building blocks.
5. Feasibility Analysis

Abstract – This chapter describes the impacts and the risks of suggested improvement. It also describes the preliminary results after implementing the solution design.

5.1 Impacts

The current software uses serialization to convert the class instantiation into a stream of bytes in order to store or transmit it to memory, a database, or a file. There is a problem with interface regarding the serialization. If we create interfaces to extract abstraction from the concrete classes, then the interfaces cannot be serialized because they do not have state. Figure 31 shows an example of the current situations that we want to change. We create an interface for the ClassY and the ClassZ, namely `IInterface`, and change the dependencies. The ClassX does not depend on the concrete classes anymore but it depends upon the interface. The changes are shown by Figure 32.

![Figure 31 – ClassX, ClassY, and ClassZ](image1.png)

![Figure 32 – IInterface, ClassX, ClassY, and ClassZ](image2.png)

However, our implementation of the solution above results in runtime exception. One of the alternative solutions is replacing the interface with an abstract class, as shown by Figure 33.

![Figure 33 – Alternative Approach of Extracting Abstraction](image3.png)
One of the negative impacts after the refactoring is that the codebase cannot be compiled. Modifying one piece of code in one place affects code in other places. Creating interfaces, breaking up classes, or using object oriented patterns can cause APIs changes in many places including the test cases. As a rule of thumb, if we improve the codebase by using refactoring strategy, then we have to compile the codebase frequently to see immediate impacts. Some tricks that could be useful are:

1. After small refactoring, a successful compilation should be ensured.
2. When changing APIs, unit tests will break. Thus, we need to design integration tests. The tests are added to ensure our changes work as expected.
3. We also need to run regression tests frequently to ensure the current behaviors do not change.

### 5.2 Risks

There is no risk of restructuring the codebase organizationally. We can move the files across the building block’s folders freely because we do not change anything in the code. However, there is one risk if we want to align the building block’s structure with physical structure. One of the improvements after we move the files from one building block to another is changing the namespace of those files from original name according to the convention, which states that the namespace should reflect the project or assembly name and the folder hierarchy. If we do this, then the risk exists that we break the current functionalities.

Changing the namespace has impact on the existing serialized files. When the software tries to read those files, a runtime exception is thrown. To avoid the risk, we could keep the namespace of serializable classes. As a rule of thumb, if we want to change the namespace, then we have to ensure the current functionalities are not broken by running the regression tests.

### 5.3 Results

Figure 34 shows the result of the implementation. There is no cyclic dependency anymore in the DSM.

![Figure 34 – DSM after Restructuring and Refactoring](image)

A building block can be aligned to a project or an assembly after we have acyclic dependencies. We can have a project or physical archive structure that reflects the organizational structure. The result shows that the solution that is designed in Section 4.1, 4.2, and 4.3 is feasible to be implemented in order to make a better structure of the codebase.
6. Future Work

Abstract – This chapter gives a brief explanation of tools that can be used to prevent the dependency problems from happening again. It also describes guidelines to understand a building block after we have cleaned dependencies.

6.1 Integrating Dependency Checker Tool

To ease the transition from the current codebase to the improved structure, some tools to check the dependency violations inside the codebase have been used. Integrating these tools in the Visual Studio can effectively prevent the dependency problems from happening again. The tools are:

1. NsDepCop
2. LiveNDepCheck

These tools are static analysis tools that allow us to enforce dependency rules in our C# projects. The tools help us to keep the (static) software architecture clean (no more unplanned or unnoticed dependencies). They are built by using Roslyn, a .NET compiler platform (CodePlex, 2015), which allow us to check the dependency violations as we type the code in the Visual Studio. These tools also can be integrated in MSBuild so that we can use dependency rules as one of build criteria.

The NsDepCop was introduced in 2012, while The LiveNDepCheck is introduced by the author during the execution of this project. The LiveNDepCheck is developed based on existing tool that is called as NDepCheck. The NDepCheck, which was originally named as DotNetArchitectureChecker, was introduced in 2010. It has proven its usefulness (and stability) in a project of 25 developers with more than two millions LOC. However, this tool has no capability to check the violations of dependency rules in edit-time. This capability is added by injecting the Roslyn component into the existing code.

Figure 35 – Class Diagram of the LiveNDepCheck
Figure 35 shows the class diagram of the LiveNDepCheck. The classes with white color come from the NDepCheck’s original codebase. They are slightly modified in order to ease the code integration. The class with green color comes from the Roslyn component. To introduce the edit-time capability in the NDepCheck, classes with blue color are added in order to integrate the Roslyn component with the NDepCheck code. The usage of NsDepCop and LiveNDepCheck is compared below.

The NsDepCop allows us to specify the namespace dependency rules only in one file per project. The file is called config.nsdepcop. A dependency is allowed if it matches an “Allowed” rule and it has no match with any of the “Disallowed” rules. We could also decide the violation severity, either warning or error, in the file. The file is written in XML format, as follows:

```xml
$xml version="1.0" encoding="utf-8">
  <NsDepCopConfig isEnabled="False" CodeIssueKind="Warning"/>
</NsDepCopConfig>
```

The example of the rules is described in Table 3.

<table>
<thead>
<tr>
<th>Line content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Allowed From=&quot;MyNamespace&quot; To=&quot;System&quot;/&gt;</code></td>
<td>MyNamespace can depend on System namespace</td>
</tr>
<tr>
<td><code>&lt;Allowed From=&quot;MyNamespace&quot; To=&quot;System.*/&quot;&gt;</code></td>
<td>MyNamespace can depend on System and any sub-namespace</td>
</tr>
<tr>
<td><code>&lt;Allowed From=&quot;MyNamespace&quot; To=&quot;.&quot;/&gt;</code></td>
<td>MyNamespace can depend on any namespace</td>
</tr>
<tr>
<td><code>&lt;Allowed From=&quot;MyNamespace&quot; To=&quot;&quot;/&gt;</code></td>
<td>MyNamespace can depend on the global namespace</td>
</tr>
<tr>
<td><code>&lt;Allowed From=&quot;&quot; To=&quot;&quot;/&gt;</code></td>
<td>Everything is allowed</td>
</tr>
</tbody>
</table>

The LiveNDepCheck allows us to specify not only the namespace but also class dependency rules in multiple files per project. The mandatory file is called DependencyRules.txt. If a dependency has no match with any rules, then a warning is emitted stating a dependency rule for the dependency has not been specified yet. The example of the contents of DependencyRules.txt file is described in Table 4.

<table>
<thead>
<tr>
<th>Line content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+ AnotherRule.txt</code></td>
<td>Include rules from the file</td>
</tr>
<tr>
<td><code>ALL := MyNamespace.*</code></td>
<td>Use ALL as abbreviation for MyNamespace.</td>
</tr>
<tr>
<td><code>(\*)\.* --&gt; \1.\.*</code></td>
<td>Define an allowed dependency: every class may use all classes from its own namespace</td>
</tr>
<tr>
<td><code>ALL ! System.**</code></td>
<td>Define a forbidden dependency: all MyNamespace classes must not use System and any sub-namespace</td>
</tr>
<tr>
<td><code>(\*)\.* --&gt; \1.\.*</code></td>
<td>Define a questionable dependency: every class may use all classes from child namespaces with warning</td>
</tr>
<tr>
<td><code>** --&gt; **</code></td>
<td>Everything is allowed</td>
</tr>
</tbody>
</table>

Both the NsDepCop and the LiveNDepCheck support the opt-out approach, that is, everything is allowed but the ones that are explicitly listed as disallowed. To implement the opt-out behavior, define an “allow all” rule and then the desired disallowed rules. If both an “Allowed” and a “Disallowed” rule are matched then “Disallowed” is the stronger. In the NsDepCop, we specify the rules as follows:

```xml
<Allowed From="\*" To="\.*"/>
<Disallowed From="MyFrontEnd.\*" To="MyDataAccess.\*"/>
```

While in the LiveNDepCheck, we specify the rules as follows:

```
** --> **
MyFrontEnd.** ! MyDataAccess.**
```
In a big project, we somehow want to skip several namespaces to be checked. We can add rules that allow unrestricted dependencies from or to a namespace. In the NsDepCop, we can specify the rules as follows:

```
<Allowed From="MyIgnoredNamespace.*" To="*"/>
<Allowed From="*" To="MyIgnoredNamespace.*"/>
```

While in the LiveNDepCheck, we also can use the above approach as follows:

```
MyIgnoredNamespace.** --> **
** --> MyIgnoredNamespace.**
```

However, the LiveNDepCheck has a feature that allows us to define an ignored rule in one line. Table 5 shows the examples.

<table>
<thead>
<tr>
<th>Line content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@MyIgnoredNamespace.*</td>
<td>Allow unrestricted dependencies from or to the MyIgnoredNamespace</td>
</tr>
<tr>
<td>@MyIgnoredClass</td>
<td>Allow unrestricted dependencies from or to the MyIgnoredClass</td>
</tr>
</tbody>
</table>

Considering the number of features that the LiveNDepCheck has, it is recommended that the LiveNDepCheck is used rather than the NsDepCop. However, the team does not want to maintain this tool. Thus, we can give suggestions of future development to the author of the NsDepCop. Several features that we want to have are:

- NsDepCop should have not only namespace, but also class dependency checking capability.
- If a dependency rule is not defined, NsDepCop should give information about the undefined rule instead of information about a rule violation.
- NsDepCop should give more elaborate description of the rule violation information such as which rule that has been violated, in which file, and in which line the rule is written.

### 6.2 Understanding a Building Block

After we have a good structure of the building blocks, the next challenge for the software developer team is to understand the building blocks itself. The architecture that is shown by Figure 1 helps us to understand the codebase in general. However, in order to understand the codebase in depth, we have to make detailed documentation for each building block. The documentation could be in the form of architectural documentation, helpful articles, diagrams, example code, and XMLDOC comments. Figure 36 shows a result of survey that was conducted by Architexa (Architexa.com, 2015). It is a small driven team of programmers who believe that better tools can make it easier to grasp code. One of the major questions that the survey had was which documentation techniques were being wanted by developers to make a codebase easy to understand. The survey was questioned to 100 developers from various Apache and Eclipse open source projects.

![Figure 36 – What Developers Want](image)
From Figure 36, we can conclude that developers want anything that can help them to understand the codebase better. However, they were lacking in documentations in the form of architecture and diagrams. While developers are getting a lot of detailed information about projects (Javadoc, examples, articles describing code use), they are lacking the high level overviews that would allow them to understand codebase at a glance. Similar condition happens in the current codebase. It lacks of documentation in the form of diagrams that can describe the codebase. Many developers utilize UML diagrams to help them in understanding code quickly and easily. Guidelines how to create those diagrams from existing building blocks by using manual reverse engineering are explained as follows:

1. **Use-case Diagram.** It is important to have functionalities overview of a building block. The use-case diagram is made to describe one or more specific building block functionalities that are used by an actor. It is almost impossible to create the diagram automatically from the codebase. The information from expert is the most important resource.

2. **Package Diagram.** A layered architecture diagram of the building block can be really helpful to know the main concepts of the building block. Currently, the architecture of a building block is not explicitly identified. We need to discover it from the existing codebase. The information from the expert is again an important resource for us in order to have an overview of the building block. Besides information from the expert, the project structure is also an important information. The folders and namespaces can be used as hints for us in order to create the architecture. By combining information from the expert and the project structure, we can discover the architecture. At first, we use Understand to analyze the dependencies between the folders in current project structure of the building block. It is expected that the output has many cyclic dependencies. By using the information from the expert and the namespace structure, we perform experiment to restructure the building block until it has no more cyclic dependency. This task can be done in relatively short of time.

3. **Provided and Required Interfaces.** By using the static analysis tool, such as Understand, we can investigate the provided and required interfaces. To investigate the external interfaces of a building block, we can use the tool to analyze the dependencies between the building block and other building blocks. By using similar approach, we also can investigate the internal interfaces of the building block. We can use the tool to analyze the dependencies between the building block’s components and investigate the provided and the required interfaces. After we have the list of provided and required interfaces, we can draw it in the component diagram.

4. **Sequence and Class Diagrams.** Sequence and class diagrams are exceptionally useful to figure out how the code works with regards to the main concepts. To generate the diagrams from the building block, sequence diagram has to be created first before the class diagram. The first step of creating sequence diagram is finding the main entry point of the building block. The entry point is usually located in the highest level package or component. At this point, we only want to create high level sequence and class diagrams since the codebase changes frequently. There is high possibility that the diagram will not be updated if there is minor change in the building block. Thus, we need to investigate the most important code, highlight it, and ignore what is not that important. It is expected that the diagrams will be relatively stable through the foreseen changes. We need to update our sequence diagram manually every time we find one important line in the codebase. In the same time, we also have to update our class diagram every time we find important class. The class diagram and the sequence diagram should be consistent. If there is a class in the sequence diagram, we should draw the same class in the class diagram. If there is an operation of a class in the class diagram, we should show the interactions between the class and other classes that use that operation in the sequence diagram. We also should not draw all classes in a single diagram.

Indeed, manual reverse-engineering is a time-consuming and tedious job. However, the output of this activity is readable and understandable.
7. Conclusions

Abstract – This chapter presents the general conclusion of the project. It summarizes the extent to which the goals have been met, aligned with the achievements of this project together with the main deliverables. Moreover, design criteria that are mentioned in Chapter 2 are revisited to investigate if they were successfully addressed.

7.1 Results

This project was created to provide a proof of concept for the modularity improvement of a large complex codebase. The result of this project is an improved version of the building block structure. The new structure has better modularity than the existing structure. Recalling from section 1.1, the goals of the project have been fulfilled considering the number of achievements of this project. In addition to that, a number of aspects were also addressed during the execution of the project. The achievements of the projects are:

1. **Codebase analysis.** Static analysis is proven to find architecture smells in the codebase. Many tools can be used to perform the analysis automatically. In this project, I used the Understand in order to find dependency problems effectively. The analysis result shows that the main problem in the codebase is the unwanted dependencies. The building blocks are involved in numerous cyclic dependencies.

2. **Codebase visualization.** The dependency graph is readable if the codebase has few dependencies. I proposed to use DSM as a more powerful abstraction to visualize the dependency problems if the codebase has a lot of dependencies.

3. **Solution design to improve the modularity.** I created an improved building block structure of codebase by using the DSM, restructuring strategy, and refactoring strategy. It is also proved that the solution is feasible to implement. As the result, I was able to create a prototype of building blocks project structure that has acyclic dependency.

4. **Recommendations for future work.** Static analysis tools also can be used to manage and monitor the dependencies. The live dependency checker tool that is integrated in the Visual Studio 2015 can ease the transition efforts. I improved the NDepCheck so that the tool can have edit-time checking capability. It can prevent the dependency problems from happening again. It also can trigger the awareness to developers if they start introducing bad dependencies. The dependency rules can be used as additional build criteria in order to keep the (static) software architecture clean. In addition to the tool, I also made experiment of understanding a building block. A guideline how to perform it and the result were also addressed.

7.2 Deliverables

Apart from this technical report, the other deliverables are:

1. Codebase analysis document. This document covers the detailed explanation of Chapter 2. It also describes analysis results which are not mentioned in this technical report.

2. Guidelines document how to break cyclic dependencies. The document covers the detailed explanation of Chapter 4 and Chapter 5. It provides step by step description how to perform certain tasks.

3. A building block document. This document shows an example of a building block documentation. The document covers the implementation of guidelines explained in Section 6.2.

4. Prototype of the codebase according to the new proposed structure.

5. Source code of the LiveNDepCheck prototype.
7.3 Open Issues
Apart from the achievements, there are also several open issues that were not fully addressed in this project. Building block documentation, which is generated by using the guidelines that are explained in Section 6.2, might be not sufficient. I possibly made mistakes on the resulting diagrams because they were generated manually. When I made the experiment to implement the guidelines, I was surprised with the amount of efforts that were required and the amount of stress that I experienced.

On the other hand, I also could not find any automatic reverse engineering tools that are smart enough to generate high level building block documentation. The output of the available tools is rather useless because the tools tend to generate every detail in the codebase. Not every detail in the codebase is important. In my humble opinion, manual reverse engineering is still considered as the best way to understand the codebase. However, there is a need to perform other investigations to find alternative ways to create more meaningful and more precise documentations from the existing codebase in the future.

7.4 Design Opportunities Revisited
At the beginning of the project, a number of criteria were chosen to be considered as important for the solution design. For each of these criteria it is verified whether the design adheres to that criterion.

- **Genericity** – I believe that the ‘genericity’ has been addressed in this project. The methods and strategies that were used in this project are very general. The approach also could be implemented in any large codebases.
- **Realizability** – The preliminary result of the project has proved that the solution design is realizable. It is feasible to make codebase more modular. During the implementation, some impacts and risks were also identified. It is very important to make sure that the improvements do not break the current functionalities. I also believe that the design solution that has been addressed is simple enough.
- **Documentation** – Considering the number of documentations that I have delivered, I believe that this criterion also has been addressed. The software development team could use these documentations as their first knowledge if they would like to start over the project in the future.
Bibliography

References


Additional Reading


About the Authors

Lindung Manik received his bachelor degree in Mechanical Engineering from Bandung Institute of Technology, Indonesia, in 2008. His bachelor thesis project tried to make a model and a software prototype of a smart lathe. It was used to simulate a conventional lathe machine in an autonomous distributed manufacturing system. After that, he joined a local company, which is called telkomsigma, as Java programmer trainee. Before he joined the Software Technology program in 2013, his last position was System Analyst. He was responsible for applications development for financial managed service solutions. He received his master degree in Information Technology from University of Indonesia in 2011.
3TU. School for Technological Design, Stan Ackermans Institute offers two-year postgraduate technological designer programmes. This institute is a joint initiative of the three technological universities of the Netherlands: Delft University of Technology, Eindhoven University of Technology and University of Twente. For more information please visit: www.3tu.nl/sai.