13TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELLING CONFERENCE, DSM'11
CAMBRIDGE, MASSACHUSETTS, USA, SEPTEMBER 14 – 15, 2011

MDM-BASED SOFTWARE MODULARIZATION BY ANALYSING INTER-PROJECT DEPENDENCIES

Alexander Mirson[1], Oleg Skrypnyuk[2], Fatos Elezi[1] and Udo Lindemann[1]

[1]Institute for Product Development, Technische Universität München, Germany
[2]Teseon GmbH, Germany

ABSTRACT

In this paper we explore the possibilities of improving software architecture by eliminating inter-project dependencies and extracting subprojects into plugins. A new approach is proposed to improve the modularization process and to support software architects to reach better decisions on how to reorganize the software system and to get loosely connected architecture in a way that the subprojects of the system are extracted into standalone plug-ins. This method is using the MDM model and has been implemented in software called LOOMEO as a standalone plugin to illustrate its applicability. As a case study we used the software LOOMEO itself to proof our concept. This method provides a solid framework for improving the refactoring process in multi-project environment.

Keywords: Software architecture, software modularity, dependency, model, matrix, MDM

1 INTRODUCTION

Bringing new software products to the market faster than the competitors has become a strategic imperative in many industrial sectors. Most of the time in the software product lifecycle is wasted on support and integration of changes in already existing code. The keys to successful maintenance of applications are a technique of extracting parts of the code and principles and methods of extracting modules of the system. Software systems with a great number of dependencies between their components restrict their further development and extension possibilities. Even the slightest changes in them have a strong impact on the whole system, since they cannot be contained (Sangal et al. 2005). Change propagation and change management in general are current topics in the research area. Sullivan et al. (2001) show the importance of the software modularity and propose an approach for evaluating software structure and modularity during the software development process and during usage of the software. This paper describes an approach to handle complex software system consisting of a large number of subprojects and to extract them into standalone plugins using dependencies between different subsystems.

One of the approaches to map the complex software system is the matrix-based model, which are being increasingly used to manage engineering systems and complex product development processes. The main benefit offered by these models is an enhanced visibility of the systems' structure, which is suitable for recognizing specific structural elements of the system and providing a holistic view of the entire system. The Design Structure Matrix (DSM) (Browning 2001) has proven to be an effective matrix-based mapping tool. Although the DSM lacks high level of detail, it illustrates complex system in a simple and useful way for both qualitative and quantitative analysis. In our approach we represent different subsystems of the software through separate domains. As DSM is limited into the analysis of one single domain, it does not take into consideration other domains and their interconnectivity.

Therefore, a more appropriate tool for managing complex systems is the Multiple-Domain Matrix (MDM). The MDM was first mentioned by Maurer and Lindemann (2007) from the Institute of Product Development at Technische Universität München. This tool is an extended version of DSM, which allows representing the structure of multiple domains at a time (Lindemann et al. 2008, 2009).

2 PROBLEM

It is almost certain that, at some point during a software system's lifecycle, there will be a need to reorganize its modularity following numerous changes to functionality and structure. Changes can

arise at any place along the software development process. Increasingly, however, change requests occur at the end of the development process. One of the main causes for implementing changes is the shortened product life cycles. Changes cost money, are in most cases time-critical and can have unexpected impact on the end product. Therefore, if a certain change occurs, the propagation of changes should be transparent, manageable and the implementation should be executed in a cost-efficient way. As we mentioned above the one way of improving the software system considering change impacts is extracting modules of the system and building loosely connected software. Software modules provide resources to other modules which are specified through the interfaces. Modularity is one of the basic principles of building software systems. In general, the software module is a single functionally completed software unit which can be identified and combined with other parts. Other benefits of using modules are the reuse of these modules in a new context and exactly mapping the structure of the system.

Different authors propose many approaches to improve software modularity. Huynh and Cai (2007) suggest an automatic approach for comparing the source code modularity and design modularity by using DSM. Huynh et al. (2008) compare source code and software structure during design phase. Arseneau and Spracklen (1994) propose a software tool for supporting the software engineers by planning the modules of the system, which uses artificial neural network applied to procedure shared information and combine it to the single modules. Sangal et al. (2005) describe different software architectural patterns in DSM and how they can be applied to improve the system structure.

The approach proposed in this paper intends to reduce complexity of splitting up software into separate plugins, thus reducing time and costs of this process. The following method aims to support this activity by assessing the involvement of each functional unit in the entire software project. This information helps software architects to make better decisions on how to organize software modules extracting them in standalone software plugins.

3 APPROACH

In this section, a step-by-step approach is going to be proposed, which will serve as a resolution guide for the research problems posed in the introduction of the paper. The ultimate goal to be achieved by applying this method is to improve software architecture by splitting up a program into separate plugins by finding relationships between the subprojects on the class level and removing them through refactoring. Methods of modular and structural analysis are used to perform this task.

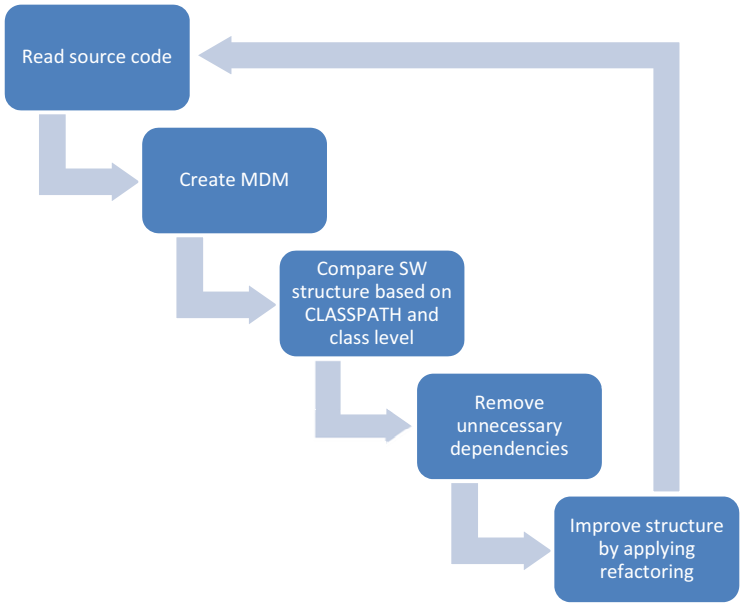


Figure 1. Steps of the MDM-based software modularization approach

The proposed method consists of the following steps, which are shown in Figure 1:

1. Read source code
2. Create MDM

3. Compare SW structure based on CLASSPATH and class level
4. Remove unnecessary dependencies
5. Improve structure by applying refactoring and go to step 1

During the first step the system is being loaded by reading the source code of different subprojects. In our use case we applied *Jar Jar Links* utility (JarJarLinks 2011) to load jar-libraries of different modules. At the next step the whole system is built up by using MDM-approach. Figure 2 shows created MDM, which represents dependencies between involved subprojects.

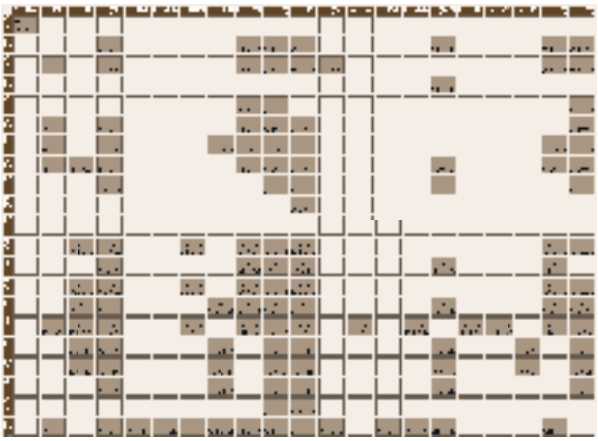


Figure 2. MDM – Overview of the whole system

The first domain is called *Projects* and contains all subprojects of the system. Its DSM depicts dependencies between subprojects on the CLASSPATH level (we applied our approach to a java-based project). These dependencies are given manually by the software developers and could be inaccurate. The causes for this problem could be:

1. Adding the maximum number of dependencies into the CLASSPATH in order to cover all passible "needs" in the future, because of using agile software development methodology, where the modelling of the whole system does not have the highest priority,
2. Lack of knowledge of the program components,
3. Dangling dependencies, which were previously required, but during the process of the program evolution are no longer needed and were not removed.

The following domains represent subprojects themselves, which are the parts of the entire system.

In this research we work on the inter-project dependencies, therefore the intra-project dependencies between classes have been omitted in this study. Thus, all attention in this paper is devoted to DMMs that represent dependencies between different subprojects.

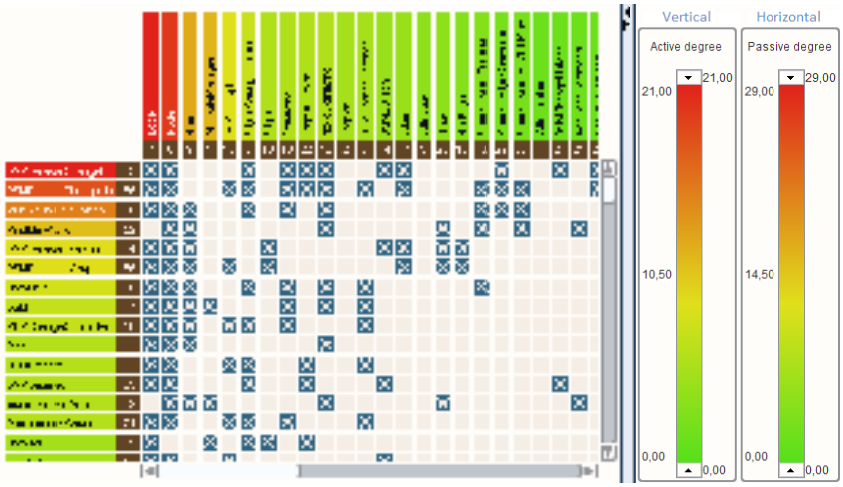


Figure 3. DMM – inter-project dependencies

We determine a dependency between two classes of different projects (an edge in DMM) when a class has at least one reference to another class. We do not take into account the closeness of classes, which

could be evaluated on the number of references to the imported class. The total number of dependencies in DMM is assigned to edge weights of the first domain's DSM. The exemplary DMM is represented in Figure 3.

Thus, comparison of dependencies on the CLASSPATH level with dependencies on the class level allows us to identify the so-called dead-dependencies. Figure 4 shows DSM with inter-project dependencies. We use this matrix to distinguish between dependencies on the different levels. If there is a dependency on the CLASSPATH level, the DSM edges are highlighted in grey. The real dependencies are displayed through the weights of the corresponding edges. As a result, all coloured edges without weights can be removed without changing the program code and any additional efforts.

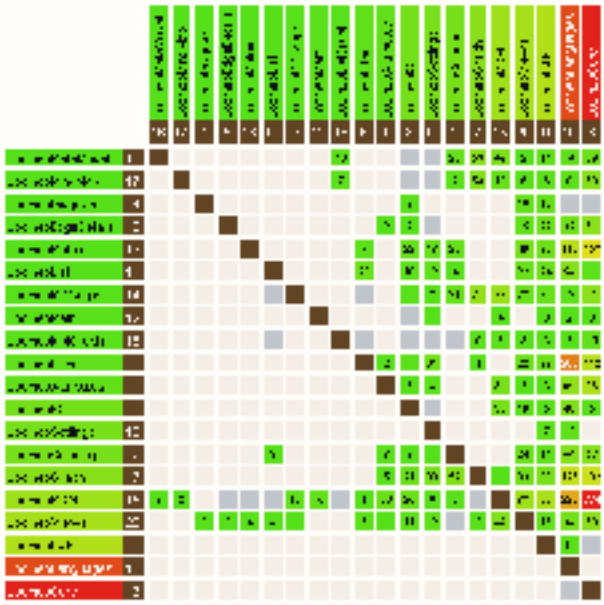


Figure 4. DSM – before modularization

Further program modularization happens through the shading of the DSM edges according to their weights. This highlighting allows us to obtain instantly a general overview of the strongly and weakly coupled parts of the system. In this way we get a qualitative view of the software system structure. The red edges with the highest weights in Figure 4 represent strongly linked subprojects, which hardly could be removed. Much more attention should be paid to the green edges with the lowest weights. They symbolize the potential for extracting program modules.

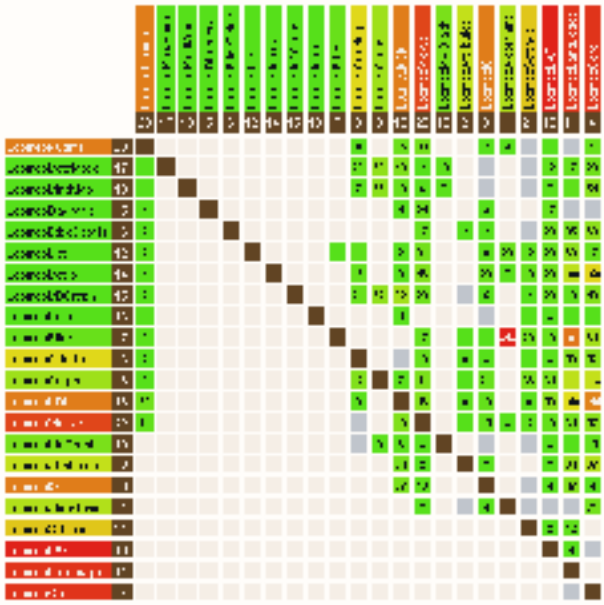


Figure 5. DSM – after modularization

Additionally, triangularization and descending sorting according to the active sum were also applied. These methods reorder the matrix in such a manner that the entries below the diagonal represent dependencies, which could be removed. Refactoring is used to reduce the number of dependencies:

1. Common variables, e.g. static variables or constants can be moved in the so-called general usage area.
2. Dependencies on the method level can be removed by moving corresponding methods into separate interfaces of a new plug-in project.

So, a new second sub-core could be identified. The first element in Figure 5 represents the new project and is strong connected with other subprojects.

At the last step subprojects that do not have any dependencies any more can be extracted into separate plug-ins.

This method is applied iteratively. After removing unnecessary dependencies a new version of the system is being reviewed. During this inspection an improvement of the system structure and occurence of new unnecessary dependencies are evaluated.

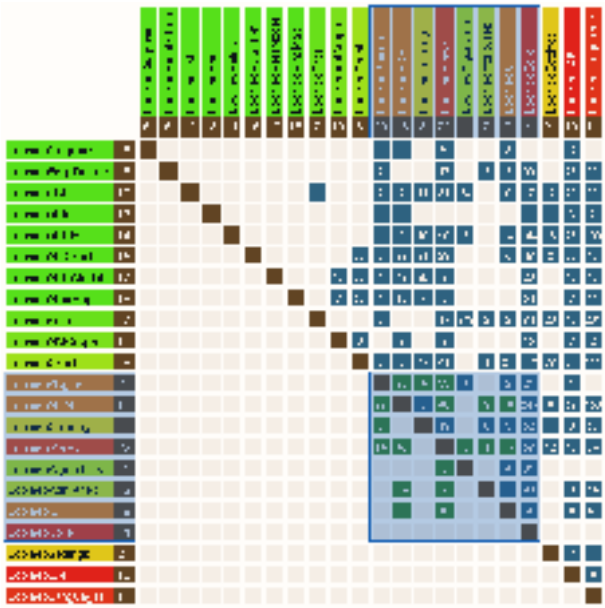


Figure 6. Identified core of the system

As a positive side effect, the core of the system can be identified by looking on double edges of the DSM shown in Figure 6. Other techniques like clustering can be applied for identifying the system core. Anquetil and Lethbridge (1999) give a brief overview of clustering techniques and their application. Wiggerts (1997) presents clustering algorithms for modularization of legacy software and uses Reverse Engineering technique.

4 CONCLUSION AND OUTLOOK

This section provides a summary of the software modularization MDM-based method, which has been presented on the previous pages.

Because of rapidly increasing complexity of software products, the need of improving their structure is raising. Well-structured software is easier to develop and to debug. It provides a set of reusable modules, which reduces the programming costs in the future.

The main advantage of the approach proposed in this paper is the illustrative idea for improving modularity of software systems by extracting subprojects into standalone plugins. This method has the following benefits:

1. Explicit representation of the system structure
2. Simplifying maintenance and modification of the software
3. New opportunities to reuse the source code

This method can be also applied recursively to analyse dependencies within the project on the class level for the package optimization. In this work we did not take into consideration the closeness of the different classes. This value could be evaluated on the number of references to the imported classes. It should be mentioned, that this method is not considered to replace the prevailing paradigm of different refactoring techniques and performs the role of extension to provide a better way for modularization of software products. However, this method provides the ability to deal with the complexity of very large software systems.

REFERENCES

Anquetil, N. and Lethbridge, T.C. (1999). Experiments with clustering as a software remodularization method. In *Proceedings of Sixth Working Conference on Reverse Engineering, IEEE,* Atlanta, GA, USA (pp. 235-255).

Arseneau, J.B. and Spracklen, T. (1994). Reengineering software modularity using artificial neural networks. In *Proceedings of WCNN* (pp. 467-470).

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

Huynh, S. and Cai, Y. (2007). An evolutionary approach to software modularity analysis. In *AcoM'07 Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques, IEEE Computer Society*, Washington, DC, USA (p. 6).

Huynh, S., et al. (2008). Automatic modularity conformance checking. In *Proceedings of the 30th International Conference on Software Engineering*, *ACM* (pp. 411-420).

JarJarLinks (2011). JarJar, http://code.google.com/p/jarjar/, accessed 19 January 2011.

Lindemann, U., Maurer, M., and Braun, T. (2008). *Structural Complexity Management: An Approach for the Field of Product Design.* Springer Verlag.

Maurer, M. and Lindemann, U. (2007). Structural awareness in complex product design – The Multiple-Domain Matrix. In *Proceedings of 9th International Design Structure Matrix Conference* (pp. 16-18).

Sangal, N., et al. (2005). Using dependency models to manage complex software architecture. *ACM SIGPLAN Notices*, 40(10), 167-176.

Sullivan, K.J., et al. (2001). The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5), 99-108.

Wiggerts, T.A. (1997). Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering, IEEE*, Amsterdam, The Netherlands (pp. 33-43).

Contact: Alexander Mirson
Institute for Product Development
Technische Universität München
Boltzmannstrasse 15
86748 Garching
Germany
Tel.: +49 (0)89 289.15121
Fax: +49 (0)89 289.15144
e-mail: alexander.mirson@pe.mw.tum.de
www.pe.mw.tum.de

MDM-Based Software Modularization by Analyzing Inter-Project Dependencies

Alexander Mirson[1], Oleg Skrypnyuk[2], Fatos Elezi[1] and Udo Lindemann[1]

[1]Technische Universität München, Germany
[2]Teseon GmbH, Germany

Technische Universität München

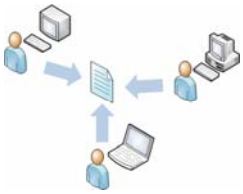
INVEST ON VISUALIZATION

Index

- Problem
- Software Development Process and Software Modularization
- MDM-based software modularization approach
- Case Study – Software LOOMEO
- Conclusion and Outlook

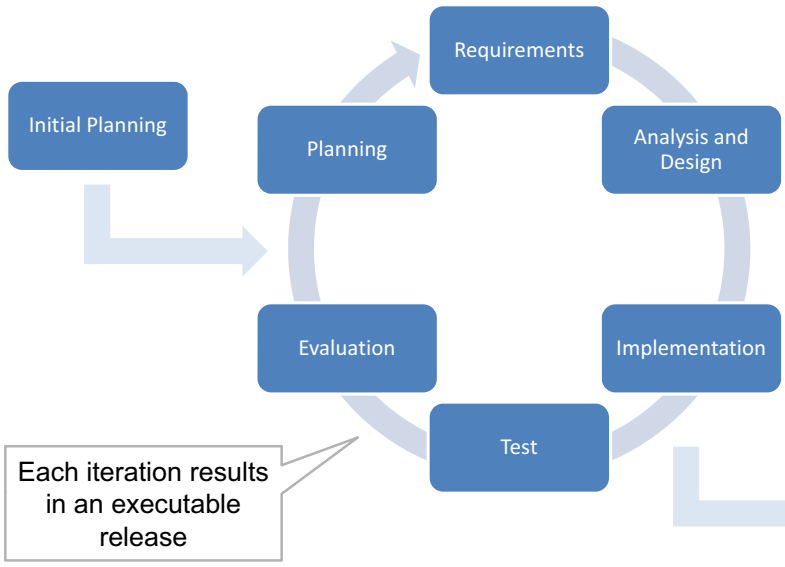
Problem

- Software systems are complex creations
  - Perform different and often conflicting objectives
  - Consist of many components
  - Many participants from different disciplines
  - Development process spans many years



- Software development projects are subject to constant change
  - Update of requirements
  - Technological changes
  - Human factor

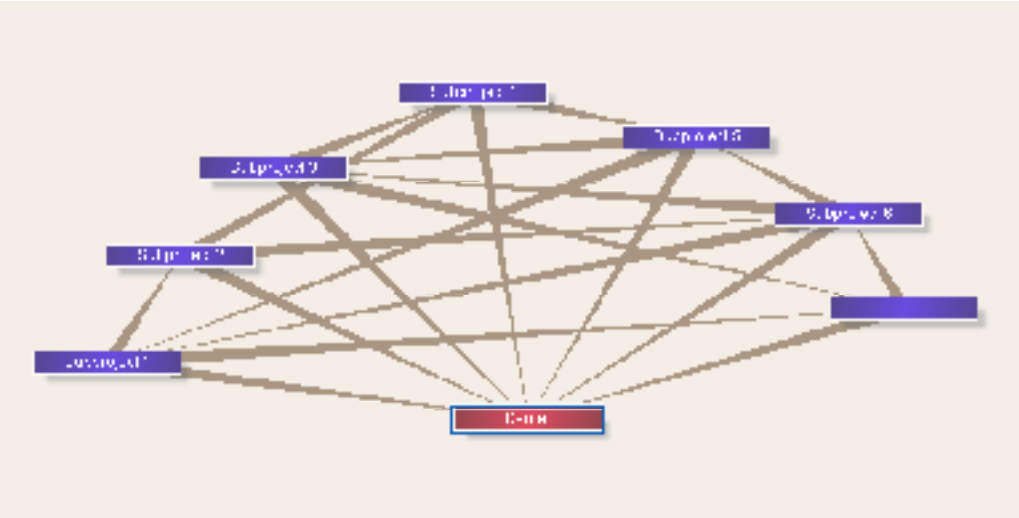
Software Development Process



- An iterative and incremental process. Adapted from Kruchten (2004)

Architectural Design and Software Modularization

- Architectural design
  - Description of a system in terms of its modules



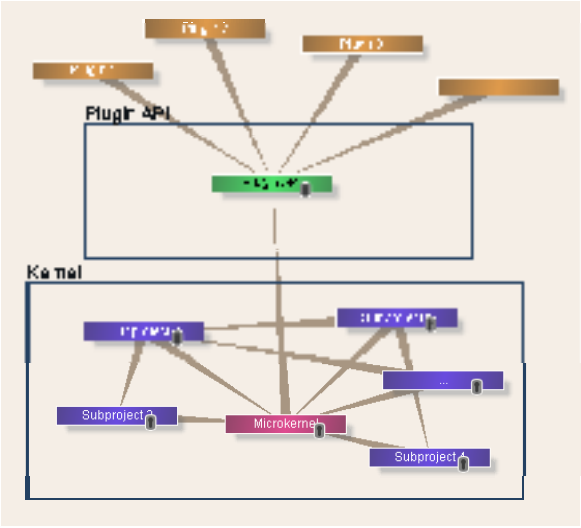
INVEST ON VISUALIZATION

Architectural Design and Software Modularization

- Objectives of software modularization
  - Loosely connected software
  - Reducing the complexity
  - Improving the software system considering change impacts
  - Flexible extension of the software

- The Law of Demeter (describes targets that are allowed for the messages within the class methods)
- Other Refactoring techniques
  - Moving features between objects
  - Organizing data
  - Simplifying conditional expressions
  - Making method calls simpler

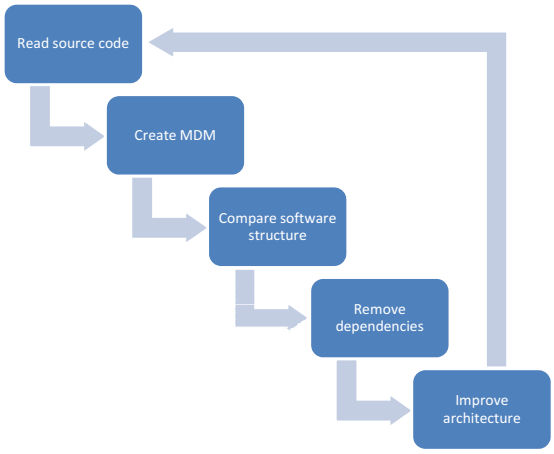
Architectural Design and Software Modularization



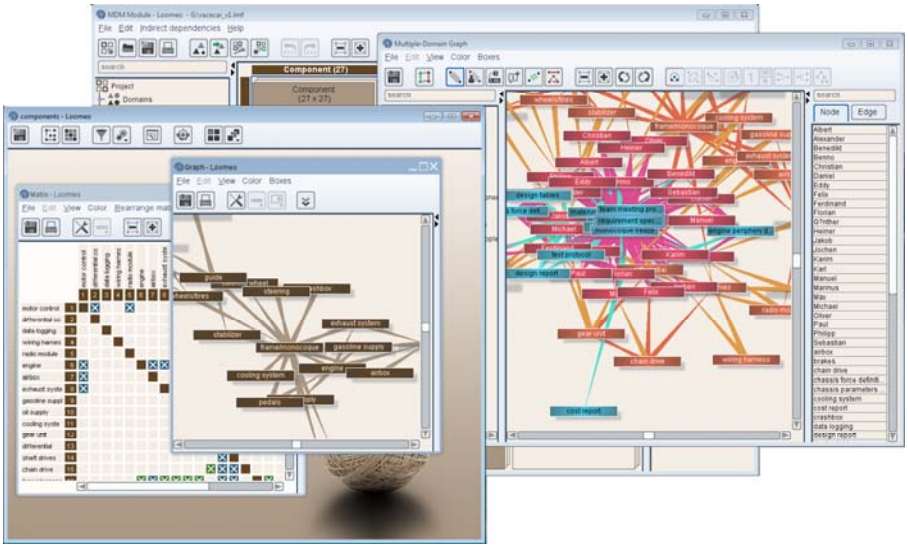
- Kernel consists of microkernel with subprojects
- Well-defined API for connecting external Plugins
- Loosely coupled Plugins

MDM-Based Software Modularization Approach

- Software modularization approach consists of the following steps:
  - Read source code
  - Create MDM
    - First DSM contains dependencies between subprojects on the CLASSPATH level
    - DMMs represent dependencies between different subprojects on the class level
  - Compare SW structure based on CLASSPATH and class level
  - Remove unnecessary dependencies
  - Improve structure by applying refactoring and go to the first step



Case Study – Software LOOMEO



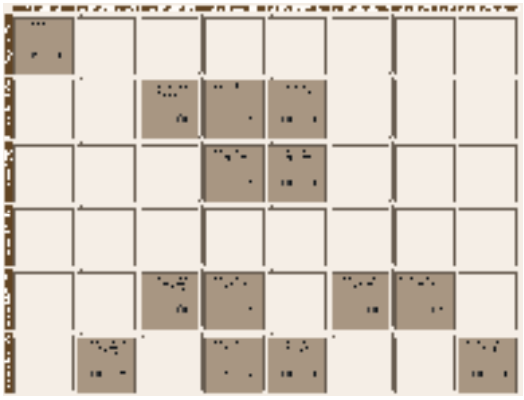
- Java-based software for visualization and analysis of network-like structures
- Over 20 subprojects, approximately 192,000 lines of code

Case Study – Read Source Code

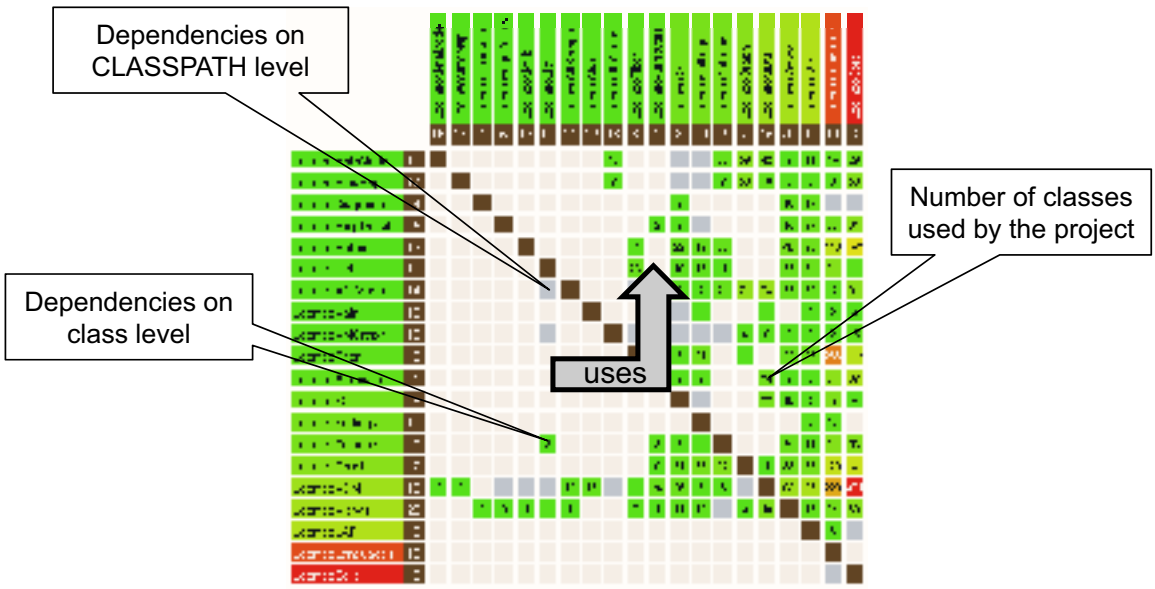
- Reading the jar-libraries of different modules using *Jar Jar Links* utility (JarJarLinks 2011)
- Collecting all dependencies between classes based on import statement without taking into account the closeness of classes

Case Study – Create MDM

- Creating MDM
- First DSM contains dependencies between subprojects on the CLASSPATH level
  - Given manually by the software developers
  - Could be inaccurate (lack of knowledge, dangling dependencies)
- DMMs represent dependencies between different subprojects on the class level

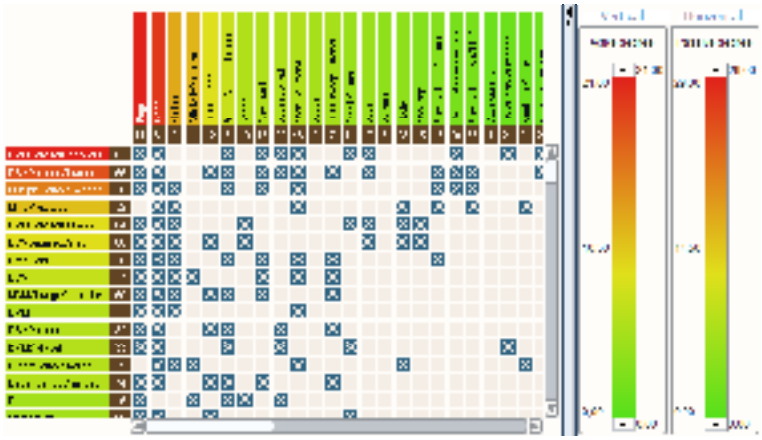


Case Study – DSM of the first domain



Case Study – Create DMM

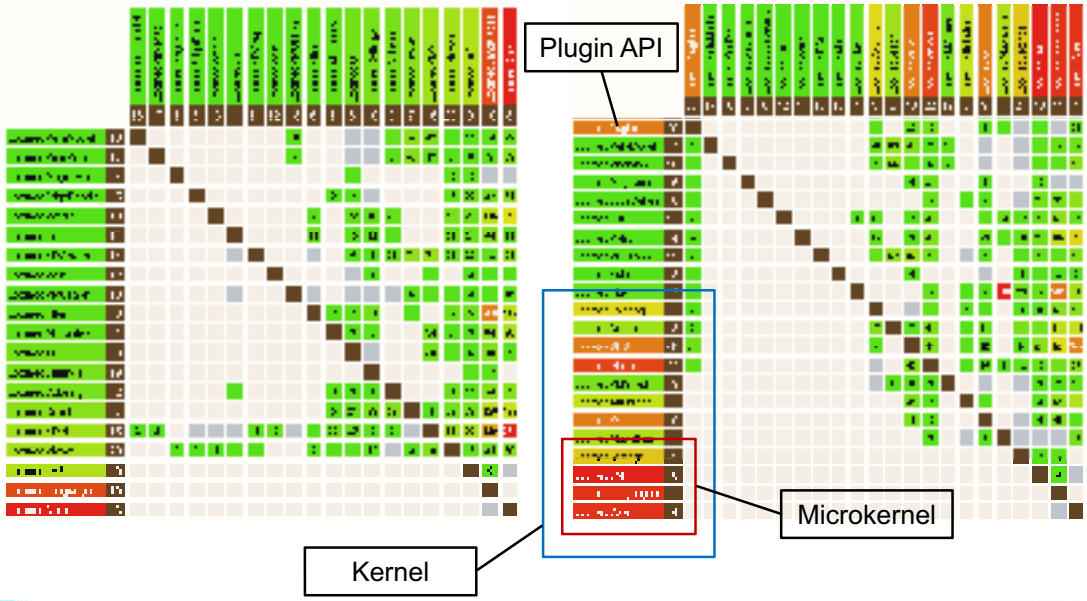
- DMM shows classes of two subprojects sorted according to active and passive sum
- Green elements represent loosely coupled classes



Case Study – Refactoring

Before applying the method

After applying the method



Conclusion

- Well-structured software is easier to develop and to debug
- It provides a set of reusable modules, which reduces the programming costs in the future

- Benefits of the approach
  - Automatic and explicit representation of the system structure
  - Simplifying maintenance and modification of the software
  - New opportunities to reuse the source code

Outlook

- Method can be applied recursively to analyze dependencies within the project on the class level for the package optimization

- Method does not take into consideration the closeness of the different classes

References

- JarJarLinks (2011). JarJar, http://code.google.com/p/jarjar/, accessed 19 January.
- Kruchten, P. (2004). *The Rational Unified Process: An Introduction*. Addison-Wesley Professional.