Towards Identification of Migration Increments to Enable Smooth Migration

Niels Streekmann[1] and Wilhelm Hasselbring[2]

[1] OFFIS - Institute for Information Technology
Escherweg 2, 26121 Oldenburg, Germany
niels.streekmann@offis.de
[2] Carl von Ossietzky University Oldenburg
Software Engineering Group
26111 Oldenburg, Germany
hasselbring@informatik.uni-oldenburg.de

Abstract. The migration of existing systems is a major problem in today's enterprises. These systems, which are often called legacy systems, are usually business critical, but difficult to adapt to new business requirements. A promising solution is the smooth migration of these systems, i.e. the systems are integrated into the system landscape and then migrated in a number of smaller steps. This leads to the question of how these steps can be identified. We propose a method based on a dependency model of the existing system and graph clustering analyses to identify these steps and define migration increments.

1 Introduction

The migration of existing systems is a recurring task in software engineering. According to [1] there are different approaches to migration, namely the migration to new environments (hardware, runtime and development) and the migration of the software architecture. We focus on the migration of the software architecture, whereby in practice this is often connected with the migration of at least one of the mentioned environment migrations.

It is assumed that the implementation of an existing system is to be migrated to a target architecture to improve its maintainability. It is further assumed that the existing system uses a structured data store, e.g. a database or XML files to manage its data. The replacement of the data store in the new system depends on certain criteria that base on the characteristics of the existing system and the circumstances of the migration project. [2] describes such criteria and corresponding migration patterns, that are derived from industrial migration projects.

The assumptions made are valid since many enterprises suffer from systems that are an important part of their business, but have become hard to maintain over the years. The reason for this phenomenon is that they have grown in years of maintenance so that the current systems do no longer respect the original architecture and data store design. This makes it hard to predict the

effect of further changes and increases the costs of future development. Another reason is that these systems often are implemented using outdated programming languages and database systems.

An important requirement for existing systems is the integration with other systems and the adaptability to new business requirements. To gain the required flexibility to fulfil these requirements many systems are migrated to service-oriented architectures (SOA). Research in that area can e.g. be found in [3], [4] and [5].

The migration to SOA can be seen as the first step of a smooth migration process, that changes the externally visible interfaces of the system, but not the architecture of the system itself. In most cases the integration with other systems is implemented using service wrappers. The achievement of maintainability is not necessarily concerned with the integration with other systems but with the integration of new functionality into the existing system. An example can be found in [6], where the existing system is wrapped by services. These are used to access functionality of the existing system from newly implemented functionality in a different runtime environment and therefore integrates the existing system with newly developed functionality.

To preserve the long-term maintainability of the integrated systems, the implementation of the systems has to be migrated to conform to its new architecture. In many cases this is only possible by reimplementing these systems. In other cases refactoring is an alternative. The decision which way is suited best depends on the quality of source code and data store design and also on the used development environment. There are several methods to perform the reimplementation. [7] gives an overview of possible approaches.

We propose a smooth migration approach. Thereby the reimplementation process is performed in a number of small steps with minimal influence on the operational systems. The systems remain operational without restrictions throughout the migration process. Another advantage is that it is possible to add new functionality or change the migration plan according to new requirements. Further reasons for smooth migration are listed in [6]. Although we concentrate on component-based reimplementation in this paper, the approach described in Section 3 could also be a preliminary step for an extensive refactoring project.

The contribution of this paper is an approach to identify migration increments that constitute the steps of a smooth migration process. The approach is based on dependency models of existing systems and graph clustering analysis. In contrast to other clustering-based migration approaches it incorporates the target architecture of the system. In this way it is prevented that a poorly structured existing system leads to an also poorly structured future system, which is the case, when the migration increments are only defined according to the dependencies in the existing system. The target architecture defines initial clusters of the existing system and thereby leads the migration process. Since the approach has not yet been evaluated in a case study it is described throughout the paper using an abstract example.

The remainder of the paper is structured as follows. Section 2 defines the smooth migration process that is the basis of the described approach. The identification of migration increments is described in Section 3. Section 4 describes related work before Section 5 concludes the paper.

2 Smooth Migration

Smooth migration represents a migration process that allows the incremental migration of existing systems. The smooth migration promises some benefits compared to other migration solutions like the cold turkey approach described in [7]. These benefits include that there is always a running system incorporating new functional requirements and that no parallel implementations are needed which would lead to higher costs.

The incremental migration process is enabled by an initial integration of the existing system into its new environment. This is similar to the migration to SOA described in Section 1. The integration is based on the future architecture of the existing system, called target architecture. We differentiate between externally visible and internally visible interfaces of the target architecture. Externally visible interfaces can be used by other system to communicate with the system while internally visible interfaces are only used by internal components of the application that is in the scope of migration. Other systems or the implementation of new functionality, as described in [6], only use the system through interfaces described in the target architecture.

The target architecture does not only describe the externally visible interfaces of the new system, but its architecture as a whole. Since the external view of the existing system conforms to the target architecture after the integration step, it is possible to migrate the implementation of the system in small steps without changing its external behaviour. The parts of the system migrated in one step are called migration increments.

To avoid inconsistencies in the collaboration of the existing and the newly implemented system during the smooth migration process, the number and size of the migration increments has to be chosen according to their dependencies. The sequence of migrating the increments also depends on the dependencies and additionally on organisational aspects and requirements which lead to new functions of the system. The implementation of this new functionality should be carried out in the new system as far as this is possible to reduce dual implementation efforts. New requirements can also lead to changes in the sequence of the migration steps.

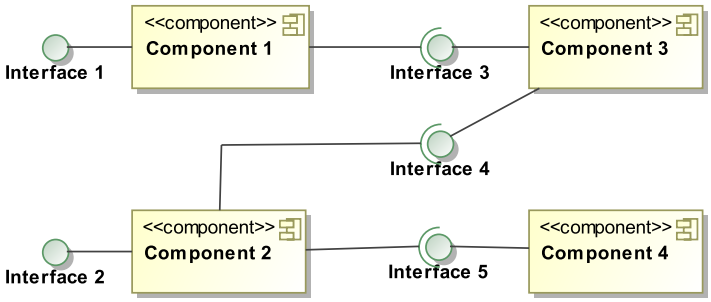
The remainder of this section describes the integration of existing systems and the migration of their implementation according to the target architecture in more detail.

2.1 Integration into the Target Architecture

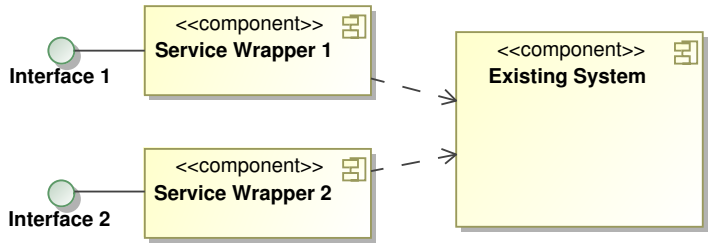
As a first step the target architecture has to be designed. This architecture represents the envisioned state of the existing system and considers maintainability and the ability to be integrated into existing or new system landscapes.

The target architecture is the source for the creation of service wrappers for the existing system. In [8] van den Heuvel describes this creation with greater detail. By wrapping the existing system it becomes reusable in its target environment. From an external point of view the existing system can now be used in the same way as its successor.

Figure 1 shows an abstract target component architecture and an existing system that is integrated using service wrappers. The interfaces of the service adapters correspond to the external interfaces of the target architecture. This target architecture is the basis for the abstract example used throughout the paper. It is shown how the existing system in Figure 1(b) implements the interfaces of the target architecture and how migration increments can be found to execute the migration of the system.



(a) Target architecture



(b) Existing system with service wrappers

Fig. 1. Integration of an existing system based on a target architecture

2.2 Migration of the Implementation

The integration into the target architecture increases the reuseability of the existing system, however the internal quality of the system and thus its maintainability is not enhanced. In the long run however this is a critical aspect for the future use and extension of the system. Hence we propose to replace the service adapters of the existing system by interfaces implemented in components of the target system in small steps.

Therefore all interfaces of the target architecture and not only the externally visible interfaces have to be mapped to functions of the existing system. Figure 2 shows the mapping of internal interfaces (1 and 2) and external interfaces (3-5) of the target architecture to functions of the existing system for the example from Figure 1 and the dependencies between the units of the existing system. Figure 3 shows an intermediate state of the smooth migration of the existing system. The interfaces 1 and 2 are already implemented in a new environment by component 1 and component 2. These still use parts of the existing system through the internal interfaces described in the target architecture.

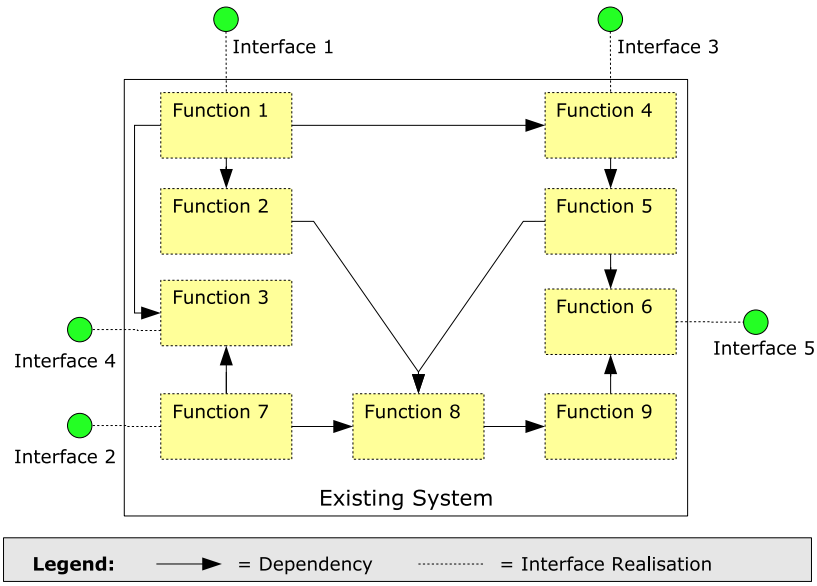


Fig. 2. Architecture model of the existing system including all target architecture interfaces

An important challenge is to find the increments of the stepwise migration, i.e. to define which parts of the existing system are replaced in the single steps. The ideal would be to migrate every interface individually. This is rarely possible in practice, since the implementation of an interface in the existing system

usually has internal dependencies that make this approach infeasible. The dependencies that are considered in the approach are described in Section 3.1.

In the following section we present an approach to identify migration increments based on dependency models. Thereby we initially focus on static functional and data dependencies. The increments are related to one or more interfaces and are characterised by minimal dependencies between the increments. The goal is to minimise the effort to resolve the dependencies between the increments in the existing code to ease the migration. In the worst case the implementation of all interfaces has to be migrated at once, which equals the cold turkey approach.

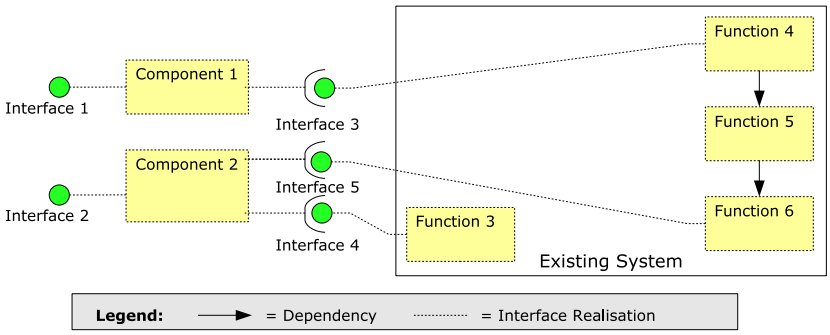


Fig. 3. Partly migrated system

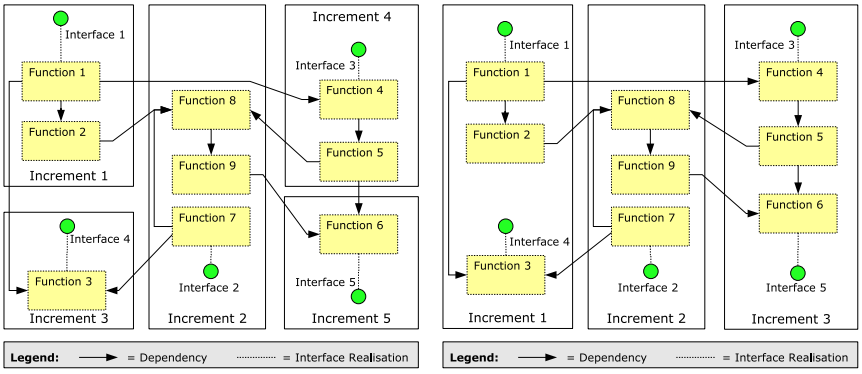
3 Identification of Migration Increments

Migration increments are the main objects in the execution of a smooth migration path. The goal is to subdivide the existing system into groups of units with high cohesion and low coupling. Units are structural entities in the implementation of existing systems, e.g. classes and methods in object-oriented programming or database tables. By doing so it is assumed that the reimplementation of the increments causes as little refactoring effort as possible. It is expected that refactorings of the existing system are necessary to enable the usage of code that has already been reimplemented in the new system.

The identification of migration increments is based on the interfaces defined in the target architecture. Implementations of these interfaces are created in the integration phase of the smooth migration process using adapters. Each migration increment has to be based on at least one interface. When a migration increment is based on an interface, it has to include all units that directly implement this interface. This includes at least all methods, functions etc. that implement an operation of the interface. Dependent units are not necessarily part of the same migration increment.

If the target implementation is component-based the ideal partitioning of migration increments probably includes one migration increment for all interfaces of one component. In this case a new component can be developed in one step and replace a certain part of the existing system. A simple example for this case is shown in Figure 4(a).

In practice however this ideal cannot always be reached. There may be internal dependencies in the existing system that can not be solved by efficient refactorings. In this case it will be more efficient to migrate the implementation of a number of interfaces in one migration increment. Figure 4(b) shows such a case.



(a) One migration increment for each interface

(b) Migration of several interfaces in one increment

Fig. 4. Partitionings of migration increments

The feasibility of a migration plan based on the migration increments depends on the dependencies between the increments. They have to be solved by a mapping to the interfaces of the target architecture or by refactorings in the existing system. To keep the effort for these refactorings as low as possible, the migration increments need to have high cohesion and low coupling. The metric for coupling and cohesion used here bases on dependencies among units. More details about the metrics are given in the next section.

3.1 Dependencies

There are manifold dependencies within a software system. Most research on dependencies concentrates on functional dependencies, as e.g. [9]. Functional dependencies are e.g. function calls, inheritance of using classes as types of variables and parameters. Besides functional dependencies data dependencies are the essential influence on the decoupling of software systems. We assume that the existing systems use a structured data store. Since there are many existing systems (also called legacy systems) with a poor and grown database design the

database is in many cases also migrated to a new design. Criteria and patterns that support the decision whether to migrate the data store or not are given in [2].

The design of the data store has a strong influence on the definition of migration increments, since interfaces implementations that work on the same data often cannot be migrated separately, because this would lead to data inconsistencies. Therefore the dependencies between code and data also have to be considered in the dependency model. In a first step we examine the dependencies between functional code and database tables. It is subject to future work to examine whether the consideration of single table columns may enhance the quality of migration increment definitions in special cases.

We defined a first dependency metamodel to describe dependency models of existing systems. The abstract classes of the metamodel are shown in Figure 5. The main elements are units and dependencies between them. Units can be code units or data units. Concrete classes of units and dependencies are omitted in Figure 5 due to readability. Examples are given in the following explanations. Code units are building blocks of imperative and object-oriented programming languages. Existing systems developed according to other programming paradigms are not considered in our approach. Examples of concrete classes of code units are function, class or method. Data units describe different ways of data representation. These can be any kind of data structures as e.g. structures in an XML file or database tables. It is also possible to define weights for dependencies. These can be used in analyses to evaluate the effort that is necessary to refactor the existing system in order to use functionality that has already been migrated to the target environment.

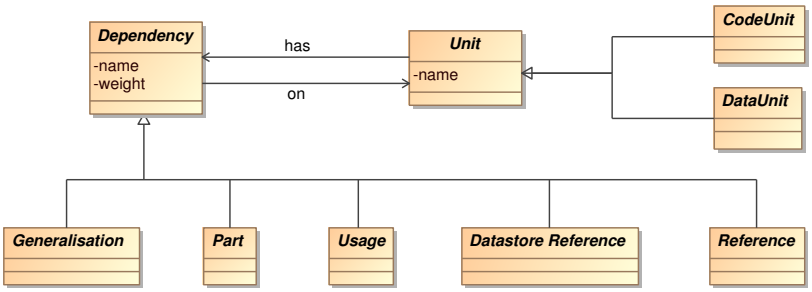


Fig. 5. Dependency Metamodel

Dependencies are relations between two code units, two data units or a code and a data unit. Figure 5 shows abstract classes of data units. Generalisation dependencies refer to concepts of object-oriented programming as inheritance and interface implementation. An example of a part dependency is the containment of methods and attributes in classes. Usage dependencies are function and

method calls as well as the usage of a certain database table. Data store references are internal dependencies of data stores as e.g. references in an XML document or a foreign key relation in a database. Reference dependencies refer to the type concept in imperative programming and cover attribute and variable types as well as parameter types.

The construction of a dependency model for an existing system is not part of the approach. However there are several methods to create such a model. Preferably the creation of the dependency model should be automated. The modelling of all dependencies of a complex system by hand is far to laborious to be applied in practice. There already are methods to automate this task, namely the analyses of the source code and the monitoring of the system at runtime. The first method requires existing analyses tools for the programming language the system is implemented in. The advantage is that no further efforts are needed, while the disadvantage is, that only static dependencies can be found. The second method requires some kind of instrumentation in order to execute the monitoring, but has the advantage that also dynamic dependencies that occur in the monitoring process can be found.

3.2 Analyses

To identify migration increments based on the dependency model, we need a suitable analysis algorithm. The analysis we propose for this purpose is graph clustering. Therefore the dependency model is transformed to a graph model. The nodes of the graph represent units while dependencies are modelled by edges. To model multiple dependencies between units or the degree of the dependencies, weights can be assigned to the edges. Weights for the degree of dependencies are also used in [9]. An extensive overview of graph clustering is given in [10].

The result of the analysis are graph clusters that represent migration increments. Figure 6 shows an exemplary graph with clusters according to the partitioning example from Figure 4(a). The units in the graph correspond to the functions with the same numbers in that figure. Since migration increments contain one or more interfaces defined in the target architecture, it has to be defined which units of the system constitute the implementation of one interface as stated above. These are necessarily part of the same migration increment. In this way the mapping between source and target architecture is addressed in the definition of migration increments.

The goal of graph clustering algorithms is to minimise the number or weights of the edges between the clusters. This corresponds to the application for the identification of migration increments, since the edges represent the dependencies between units and therefore minimising connections between clusters corresponds to the minimisation of coupling and maximisation of cohesion in migration increments.

To incorporate the link to the target architecture into the algorithm that defines clusters on the dependency model of the existing system, we propose to define initial clusters on the basis of the interfaces of the target architecture.

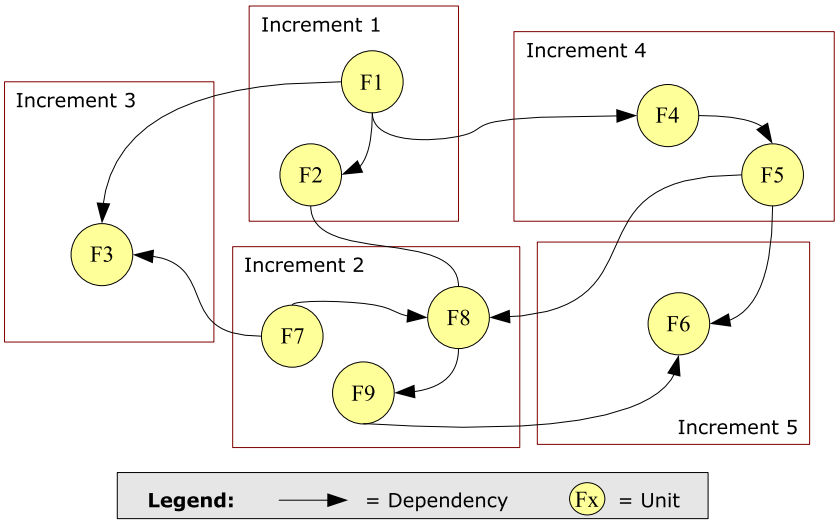


Fig. 6. Graph clustering example

These clusters include all units that directly implement the interfaces. The clustering algorithm will then add the remaining units of the existing system to the clusters based on the dependencies they have to the initial clusters. In a second step it has to be checked whether the dependencies between two clusters can be solved. If the refactoring effort is to high it should be taken into consideration to migrate the clusters as one migration increments as shown in Figure 4(b).

The identified migration increments are the basis of the smooth migration path of the existing system. The implementation of new components will replace the according units of the existing system, grouped in the migration increment that implements the same interfaces as the new component. The order of the reimplementation of migration increments is based upon the dependencies between the migration increments and new functional requirements that need to be fulfilled by the system. New requirements should preferably be implemented in the new runtime environment to reduce the effort of repeated implementations. This corresponds to the Dublo pattern described in [6].

4 Related Work

The research project Application2Web [11] has the goal to migrate existing systems to modern web applications. These can be seen as a subset of the more general goal to migrate existing systems to new architectures described here. The project uses graph clustering based on static dependencies between code units to identify the components of the new system. Therefore the basic concept is the same as in the approach described here. The difference is that components of the target architecture and their interfaces are already given in our approach

and that our goal is to find ideal migration increments for a smooth migration process.

[12] and others describe methods for the transformation of software systems, whereby transformation means the automated transfer into a new environment. Our approach assumes the reimplementation of software systems instead. The reason is that we concentrate on existing systems that lack a high quality software architecture that allows for the long-term usability of the system. The main goal of the approach is to migrate existing systems to that kind of architectures.

5 Conclusions and Future Work

The paper introduces an approach to identify migration increments in a smooth migration process. The identification is based on a target architecture and a dependency model of the existing system describing the dependencies between units in the system. The units are grouped in migration increments using graph clustering algorithms. These migration increments are the building blocks of a smooth migration path.

In our future work we will examine whether the Knowledge Discovery Metamodel [13] is suitable to replace our own dependency metamodel. This would conform our approach to more standardised methods and would ease the integration with reengineering tools that also support the KDM. The goal of the KDM is to define a common representation of existing systems for the integration between analysis tools.

Furthermore suitable graph partitioning algorithms have to be found for the implementation of the approach and transformations have to be defined from dependency and target architecture models to input models of the algorithms. More work also has to be done on the mapping from analysis results to a smooth migration plan.

The approach is going be evaluated in case studies. The goal of these studies is to find out whether the graph clustering algorithms executed on dependency models lead to suitable migration increments of existing systems that fit the target architecture. Furthermore they will show which degree of dependencies in an existing system is acceptable in order to migrate it in a smooth migration process and from which degree on it is more reasonable to migrate the system using another migration method.

References

1. Gimnich, R., Winter, A.: Workflows der Software-Migration. Softwaretechnik-Trends **25**(2) (2005) 22–24
2. Hasselbring, W., Büdenbender, A., Grasmann, S., Krieghoff, S., Marz, J.: Muster zur Migration betrieblicher Informationssysteme. In: Tagungsband Software Engineering 2008. Lecture Notes in Informatics, München, Köllen Druck+Verlag (February 2008)

3. Ziemann, J., Leyking, K., Kahl, T., Werth, D.: Enterprise Model driven Migration from Legacy to SOA. In Gimnich, R., Winter, A., eds.: Workshop Software-Reengineering und Services. Fachberichte Informatik, Koblenz, Germany, University of Koblenz-Landau (2006) 18–27
4. Winter, A., Ziemann, J.: Model-based Migration to Service-oriented Architectures - A Project Outline. In Sneed, H., ed.: CSMR 2007, 11th European Conference on Software Maintenance and Reengineering, Workshops, Vrije Universiteit Amsterdam (3 2007) 107–110
5. Sneed, H.M.: Migration in eine Service-orientierte Architektur. Softwaretechnik-Trends **27**(2) (May 2007) 15–18
6. Hasselbring, W., Reussner, R., Jaekel, H., Schlegelmilch, J., Teschke, T., Krieghoff, S.: The Dublo Architecture Pattern for Smooth Migration of Business Information Systems: An Experience Report. In: Proceedings of the 26th International Conference on Software Engeneering (ICSE 2004), IEEE Computer Society Press (May 2004) 117–126
7. Brodie, M.L., Stonebraker, M.: Migrating legacy systems: gateways, interfaces & the incremental approach. Morgan Kaufmann Publishers Inc. (1995)
8. van den Heuvel, W.J.: Aligning Modern Business Processes and Legacy Systems - A Component-Based Perspective. MIT Press (2007)
9. Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion In Object-Oriented Systems. In: Proceedings of the 3rd International Symposium on Applied Corporate Computing (ISACC1995). (Oct 1995)
10. Schaeffer, S.E.: Graph Clustering. Computer Science Review **1**(1) (2007) 27–64
11. Andriessens, C., Bauer, M., Berg, H., Girard, J.F., Schlemmer, M., Seng, O.: Strategien zur Migration von Altsystemen in komponenten-orientierte Systeme. Technical report, Fraunhofer IESE / FZI Karlsruhe (2002)
12. Kühnemann, M., Rünger, G.: Modellgetriebene Transformation von Legacy Business-Software. In: 3. Workshop Reengineering Prozesse (RePro2006),Software Migration. Number 2 in Mainzer Informatik-Berichte (2006) 20–21
13. Gerber, A., Glynn, E., MacDonald, A., Lawley, M., Raymond, K.: Knowledge Discovery Metamodel - Initial Submission, OMG Submission admtf/04-04-01 (2004)