REPRESENTING DEPENDENCIES
AND
DEPENDENCY STRUCTURE MATRIX
BASED VISUALIZATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ÇANKAYA UNIVERSITY

BY

OĞUZHAN YÜCETÜRK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING
SEPTEMBER 2010

Title of the Thesis: **Representing Dependencies and Dependency Structure Matrix Based Visualization**

Submitted by **Oğuzhan YÜCETÜRK**

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr.Taner ALTUNOK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr.Mehmet R. TOLUN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Ersin ELBAŞI Co-Supervisor	Prof. Dr.Mehmet R. TOLUN Supervisor

Examination Date : September, 24 2010

Examining Committee Members:

Prof. Dr.Mehmet R. TOLUN	(Çankaya Univ.)	
Asst.Prof.Dr. Reza HASSANPOUR	(Çankaya Univ.)	
Asst.Prof.Dr.Kasım ÖZTOPRAK	(Karatay Univ.)	
Dr. Ersin ELBAŞI	(TUBITAK)	

STATEMENT OF NON-PLAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name	: Oğuzhan Yücetürk
Signature	:
Date	: 24.09.2010

ABSTRACT

REPRESENTING DEPENDENCIES AND DSM BASED VISUALIZATION

Yücetürk , Oğuzhan
M.Sc., Department of Computer Engineering

Supervisor: Prof. Dr. Mehmet R. Tolun

Co-Supervisor: Dr. Ersin Elbaşı

September 2010, 59 pages

This work presents an approach to general definition and identification of the types of dependencies with the object oriented perspective. Also the Unified Modeling Language (UML) way of dependency representation is investigated. In the second part of the thesis the dependency structure matrix (DSM) methodology is introduced and aspects of DSM are shown. For the purpose of representing dependencies a basic application is developed. The application uses main partitioning algorithms for the implementation of DSMs.

Keywords: Dependency Representation, Object Oriented Programming, UML, DSM, Algorithm

ÖZ

BAĞIMLILIK GÖSTERİMİ VE DSM TABANLI GÖRSELLEŞTİRME

Yücetürk , Oğuzhan
Yükseklisans, Bilgisayar Mühendisliği Anabilim Dalı

Tez Yöneticisi: Prof. Dr.Mehmet R. Tolun

Ortak Tez Yöneticisi: Dr. Ersin Elbaşı

Eylül 2010, 59 sayfa

Bu çalışmada, genel bağımlılık tanımı ve bağımlılık tipleri belirlenmesi nesne tabanlı perspektif yaklaşımıyla sunulmuştur. Ayrıca bağımlılığın UML gösterimi incelenmiştir. Tezin ikinci bölümünde DSM metodolojisi takdim edilerek, bu yöntemin değişik yönleri gösterilmiştir. Bağımlılık gösterimi amacıyla bir uygulama geliştirilmiştir. Belirtilen uygulama, DSM gerçekleştirmesi için temel bölümleme algoritmalarını kullanmaktadır.

Anahtar Kelimeler: Bağımlılık Gösterimi, Nesne Tabanlı Programlama, UML, DSM, Algoritma

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor Prof. Dr. Mehmet R. Tolun and co-supervisor Dr. Ersin Elbaşı for their guidance, advices, criticism, encouragements, and insight throughout the research.

TABLE OF CONTENTS

STATEMENT OF NON-PLAGIARISM iii
ABSTRACT iv
ÖZ v
ACKNOWLEDGMENTS vi
TABLE OF CONTENTS vii
LIST OF TABLES ix
LIST OF FIGURES x
LIST OF ABBREVIATIONS xi
CHAPTERS:
1.INTRODUCTION 1
1.1 Background 1
1.2 Motivation 2
1.3 Objective 3
1.4 Thesis Outline 3
2.THEORETICAL BACKGROUND 5
2.1 Dependency Features 5
2.1.1 Definition of Dependency 5
2.1.2 Classification of Dependency 7
2.1.3 Attributes Which Describe a Dependency 9
2.1.4 Dependency Type Hierarchy 11
3.OBJECT-ORIENTED PROGRAMMING 12
3.1 Object-Oriented Software Engineering 12
3.1.1 Paradigms 12
3.1.2 The Four Principles of Object-Orientation 13
3.1.3 Objects, Components, Patterns, Architectures, Frameworks 14
3.2 Object-Oriented Dependency 15
3.2.1 Software Component Ontology 18

4.DEPENDENCIES IN UML 20
4.1 Object-Oriented Dependency 20
4.2 The Unified Modeling Language (UML) 22
4.2.1 Concepts of the UML 2.0 22
4.2.2 Dependencies 24
4.2.3 UML Characteristics With Respect To Dependency 28
5.DESIGN STRUCTURE MATRICES 30
5.1 Basic Concepts 30
5.2 Dependencies 31
5.3 Types of DSMs 33
5.4 Roles of DSMs 36
5.4.1 Project Management Tool 37
5.4.2 System Analysis Tool 37
6.DSM-BASED VISUALISATION 38
6.1 Reading, Partitioning, Clustering 38
6.1.1 Reading a DSM 38
6.1.2 Partitioning a DSM 39
6.1.3 Clustering a DSM 40
6.1.4 Numerical DSMs 41
6.2 Representing Dependencies Using DSMs 43
6.3 Related Work 47
6.4 Lattix LDM 47
6.5 NDepend 49
6.6 DeMatrix 49
7.DSM INTERFACE IMPLEMENTATION 51
7.1 Algorithms 51
7.2 DSM Interface Component 56
8.CONCLUSIONS AND FUTURE WORK 58
8.1 Conclusions 58
8.2 Future Work 59
REFERENCES R1
APPENDIX A1
CURRICULUM VITAE A1

LIST OF TABLES

Table 2-1 Dependency Attributes 10
Table 3-1 Types and Relations Defined in SCO 18
Table 4-1 UML Relationship Types [13, 14] 24
Table 4-2 Kinds of Dependencies 26
Table 5-1 Four different types of data that can be represented in a DSM [33] 35
Table 6-1 Importance Ratings 42

LIST OF FIGURES

Figure 2-1 Multidimensional Space of Dependencies [16].. 8
Figure 2-2 Dependency Type Hierarchy [6] 11
Figure 4-1 Example of Class and Object 23
Figure 4-2 Dependencies... 28
Figure 5-1 Sequential relationship (Dependent) 32
Figure 5-2 Parallel relationships (Concurrent)... 32
Figure 5-3 Coupled relationships (Interdependent) 33
Figure 5-4 A taxonomy of DSM types according to Browning[32, 33] 34
Figure 6-1 Sample DSM 39
Figure 6-2 Example of Partitioning 40
Figure 6-3 Example of Clustering.. 41
Figure 6-4 Sample Packages 44
Figure 6-5 Sample DSMS Package Level.. 45
Figure 6-6 Sample DSMS Class Level 46
Figure 6-7 Sample DSMS Method Level... 46
Figure 6-8 Lattix LDM Eclipse plug-in showing a DSM for Apache Ant 47
Figure 6-9 DeMatrix applet displaying a DSM for the source code of JAPAN 50
Figure 7-1 An example directed graph and its corresponding adjacency matrix....... 52
Figure 7-2 Reachability matrix 53
Figure 7-3 Original DSM 57
Figure 7-4 Optimized and partitioned DSM 57

LIST OF ABBREVIATIONS

CQL	Code Query Language
DSM	Dependency Structure Matrix
DSM	Design Structure Matrix
JAPAN	Java Package Analyzer
NDEPEND	Microsoft .Net Dependency Management Tool
NDSM	Numerical Dependency Structure Matrix
LDM	Lightweight Dependency Model
OMG	Object Management Group
OO	Object Oriented
SCO	Software Component Ontology
UML	Unified Modeling Language

CHAPTER I

INTRODUCTION

This chapter gives a short introduction to the thesis. First, some background information about the software dependency concept is provided. Then motivation for the thesis and the objective will be given. Also outline of the thesis will be discussed.

1.1 Background

Several smaller elements are put together for building software systems. In an excellent model, each element (component, object, function i.e. any subsystem or code unit) works clearly with its defined functionality and in harmonization with the other parts of the whole system.

Usually, none of the parts of the system creates the functionality directly. Rather, each part contributes an aspect of the functionality and depends on other parts to provide other aspects. However, each new element complicates the system structure, making it harder to document and test. [9]

Since, software development architecture models are evolving from simple programming units to high level complex systems, the identification of dependencies becomes increasingly important. Dependency analysis has many applications in software engineering activities including software understanding, design, development, testing, debugging, and maintenance.

Many principles and techniques for managing software dependencies have been developed to control and increase the changeability of the software. The key benefit these techniques and tools give software that is more clear, better formed and easier to maintain and debug.

1.2 Motivation

"For want of a nail, the shoe was lost.
For want of a shoe, the horse was lost.
For want of a horse, the rider was lost.
For want of a rider, the battle was lost.
For want of a battle, the kingdom was lost,
And all for the want of a horseshoe nail."

Dependency management is essential for a couple of reasons. The original purpose of dependency analysis is to better ensure the quality of software, and in particular warn about possible structural problems early on. Dependency management can also facilitate other management activities such as fault management, accounting management or tailorability management. [1]

Dependency management is fundamental because it directly impacts the changeability and testability of the system. A system where many subsystems are co-dependent quickly becomes difficult to change because every modification potentially requires rebuilding and retesting of all the subsystems. [9]

Sound dependency management supports coordination and communication among involved parties. It requires that efforts are made to know and to keep track of dependencies, throughout the lifecycle of a software product. Moreover, it means that opportunities for improved modularization, resulting in dependency minimizations are detected, evaluated and pursued at various development stages. [26]

In summary, it is of vital importance that developers know well what their code relies on and how and why it does so. This emphasizes the importance of dependency management for software development.

1.3 Objective

The objectives of this thesis are twofold. The first goal of this thesis is to investigate the dependency concept from the perspective of software engineering and defining the concept of dependency and, secondly, exploring dependency structure matrices (DSM) for the purpose of analysis of software products.

DSM-based support tools for software development offer new opportunities because DSMs communicate information on prevailing dependencies in a clear and concise way. Furthermore, it is expected that DSM visualizations can facilitate the identification of opportunities for dependency minimizations. [26]

1.4 Thesis Outline

The remainder of the thesis is structured as follows:

Theoretical definitions of concept of dependency will be given in chapter 2. The chapter is intended as introduction for basic concepts and classification of dependency concept.

Chapter 3 gives an introduction to theories behind Object-Oriented Software in general. In this chapter the thesis is focused on OO dependency and describes some of the different forms it can take.

Chapter 4 introduces briefly the benefits of modeling and the role of Unified Modeling Language (UML) in conceptual design. UML semantics of relationships with the representations of dependency types will be defined.

Chapter 5 presents a thorough introduction to DSM. The DSM literature and basic concepts for DSM will be defined. Also, DSM types and main roles will be investigated.

Chapter 6 elaborates DSM visualization topics and some examples for the simplicity of understanding the DSM concept will be given. A couple of different software tools for DSM will be described. It is not an extensive tool evaluation, but rather a brief summary of tools will be given.

Chapter 7 implements an application to support the visualization of DSMs and optimization of system dependencies by using the partitioning algorithm.

In Chapter 8, the thesis ends with the summary and conclusions part including important conclusions from this study. Finally, the future possible work related with this study will be given.

CHAPTER II

THEORETICAL BACKGROUND

This chapter consists of theoretical definitions of concept of dependency and its associated terms. Related literature which will introduce the basic concepts and theoretical views relevant for the thesis are presented. Here is a brief overview of the chapter: First, main features of the dependency concept will be analyzed. In the subsections the definition, classification, attributes and hierarchy of the dependency concept will be extended.

2.1 Dependency Features

The aim of this section is to discuss the main characteristics of the concept of dependency.

2.1.1 Definition of Dependency

There is much related work concerning concept of dependency, and a wide range of mechanisms have been proposed to analyze the dependencies of entities in a system. [31]

In the present literature, an actual definition and characterization of a dependency is usually avoided, and it is difficult to separate the discussion of the dependency from the particular domain of interest. [5]

In the dictionary the meaning of words dependence and dependent are given as follows [12]

"Dependence 1: the quality or state of being dependent; esp.: the quality or state of being influenced or subject to another ..."

"Dependent 2 a: determined or conditioned by another: contingent b: relying on another for support c: subject to another's jurisdiction ..."

Cox, et. al. [5, 6] starting from this point, proposes an English language definition of a dependency, influenced by the view that is interested in observable attributes:

A *dependency* is a relationship involving two or more elements where a change of state in one or more elements leads to a potential for a change of state in one or more other elements.

Dependencies are ubiquitous in a computer system. As an example, consider an Internet browser used in a typical desktop computer. Several dependencies associated with it can be identified [6]:

- Access to a web server depends on the network hardware in the computer, application layer support, and Internet services;
- The operation of a picture viewer is dependent upon security settings (e.g., picture files must have read permission);
- The browser is dependent upon the operating system for various services such as saving files, obtaining fonts, etc.
- The operating system is dependent upon a mouse and keyboard for its input

Interoperability and intercommunication is now being provided by different technical specifications and disciplines such as networks, software development, application integration, and human to machine interfaces. [31]

Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these

components. Such a system may in turn be used as an element in a larger system design. [18]

Usually, a group of components depend on each other in order to supply complex system functionality. Any modification to a component can cause the change of composite functionality. In addition, the replacement of a new version component will also cause the change of dependency between components. [20]

2.1.2 Classification of Dependency

The second activity is the classification of dependencies. In order to compare the different approaches towards classification of dependencies, specific research papers will be analyzed in detail.

In the paper “The coordination of dependencies” the authors mention various types of dependencies and their corresponding coordination process. They define the coordination of dependencies as a management task. The analogy is made to a well-coordinated basketball team. The team moves transparently in a coordinated fashion, and the players seem to move automatically, but their movements are inter-related. What one member of the team does or does not do affects the team as a whole. [30]

In human systems, hierarchical dependencies are found in the way people organize their activities, such as in an enterprise. In computer systems, many examples of dependencies such as in distributed systems can be seen. They are in a hierarchy of dependent modules so the system can collaborate properly.

In [1] authors presented the different occurrences of dependencies between components in distributed applications. The first group comprehends syntactical dependencies, where a communication between two components actually takes place (Event Flow, Data Flow, and Functional Dependency). The second group is

also based on a syntactical level, but without direct communication between the dependent components (Implicit Dependencies). The latter group of dependencies finally describes semantic properties between components.
Keller, et. al. [16] states that since dependencies come in different flavors and have varied characteristics, dealing with them in a systematic way can be facilitated by classifying them into groups with similar properties. According to Keller, et. al. a dependency, from the viewpoint of classification, has the following characteristics in Figure 2-1:

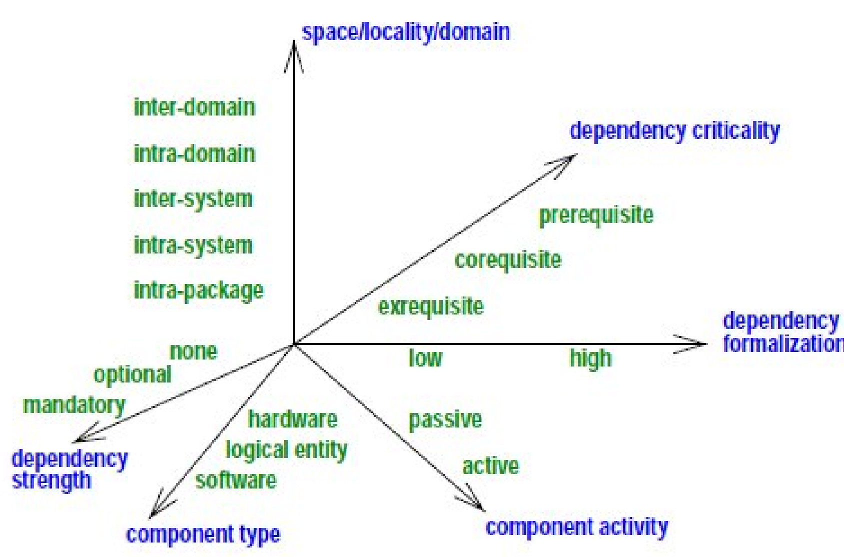


Figure 2-1 Multidimensional Space of Dependencies [16]

In [19], the authors introduces the forms of software dependencies as *Source code dependencies* occur when distinct packages rely on the same original code, usually through files. Such dependencies may not be explicitly manifested at run-time; however, they are important since changes to the code still risk breaking the dependent packages. A common and desirable example of a pure source dependency without an explicit run-time dependency is the use of abstract interfaces.

Binary dependencies occur when a program or a library depends on another library.

Embedded dynamic dependencies are made common by certain programming models: object factories, embedded interpreters, dynamic loading, and configurable internal services for example. These are of course mostly good design and decrease coupling. The flexibility becomes a dependency when a client package uses hard-coded literal strings (or other such constants) in its use of these services, such as loading an explicit library by name. In functional terms and from the testing point of view, the source is as inflexible as if the dependency was a more conventional source dependency, and ought to be considered as such in the examination of that package.

Knowledge dependencies occur when one part of the program makes assumptions about what the other parts do. Common forms are read/write data dependencies or communication through shared data. Multi-threaded applications have many other synchronization issues. Assumptions about structure layouts, object sizes, virtual function table layouts and those made when in lining code are also implicit knowledge. If there is too much such knowledge in many places, the system may become fragile.

In [23] author states that a software system may be structured in many different ways depending on the problem domain, design methodology, and implementation environment. But almost all software systems have components that are identifiable as data items, data types, subprograms, or source files.

2.1.3 Attributes Which Describe a Dependency

In [5, 6], the approach aims at discovering and focusing on the very ambitious attempt to produce what might be called an ontology of dependencies. This includes both the identification of a set of attributes which apply to every dependency and the development of a general dependency type hierarchy based upon those attributes.

Attributes are shown with general definitions in Table 2-1. [5, 6, 16]

Table 2-1 Dependency Attributes

Feature	Definition	Possible Values
Need	The need on which the dependency is based. Is usually represented as a list of required capabilities.	Authorization Resources Provided Testing
Criticality	A measure of the importance of this dependency to the success of the "dependent" entity.	Not Applicable High Medium Low
Frequency	A measure of the frequency of the need/criticality – how often does the need/importance influence operation? A numeric value representing how often the dependency exists during a particular time period.	Daily Hourly Yearly
Impact	Possible repercussions of failure at this dependency.	None Mission Compromised Information- Unreliable Performance- Degraded Corruption/Loss of Information/ Communication
Sensitivity	How vulnerable is this dependency to compromise or failure?	Fragile Moderate Robust
Stability	A measure of the continuity over time of the dependency's vulnerability to compromise or failure. One way of looking at stability is to ask the question: "When is the dependency fragile/moderate/*etc*?"	Very-Stable Infrequent Periodic Certain-Defined- Times
Cardinality	How many entities are involved?	Independent Binary N-ary
Direction	How are the entities involved?	Symmetric Anti-symmetric Asymmetric

2.1.4 Dependency Type Hierarchy

Once a complete set of dependency attributes is identified, it will then be possible to establish a type hierarchy, resembling a lattice, based upon those attributes and their values. Using this hierarchy, specific types of dependency are characterized and related to each other, and dependency types can be chosen to be applicable to particular domains. Eventually, it should be possible to fully populate the dependency type hierarchy based upon the attributes identified. [5]

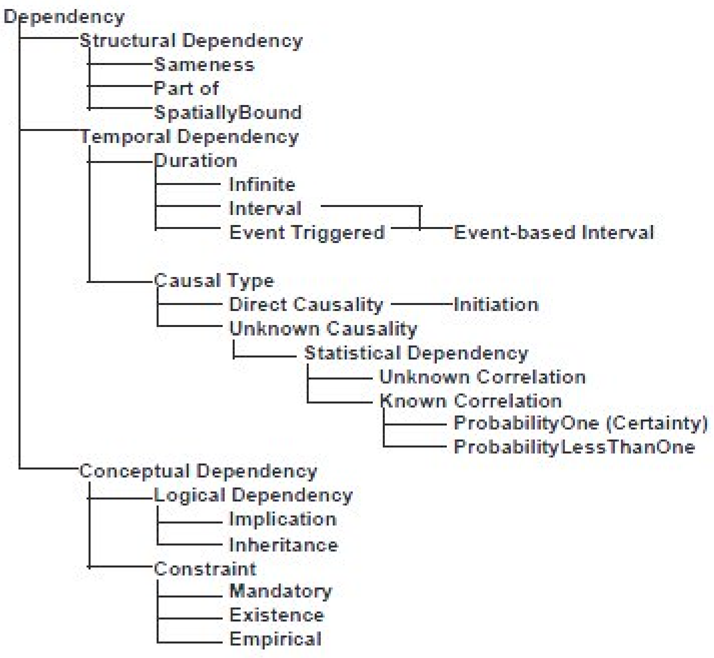


Figure 2-2 Dependency Type Hierarchy [6]

CHAPTER III

OBJECT-ORIENTED PROGRAMMING

In this chapter, state-of-the-art methodologies from Object-Oriented Software Engineering and analyze the dependency concept of OO programming will be reviewed. An overview of the paradigms of current object-oriented methodologies will be provided. OO dependency will be introduced and described some of the different forms it can take. The way of how dependency is expressed implicitly and explicitly and the usage of dependency will be analyzed. Also, the ontology development methodology for Software Component Ontology will be defined in terms of the object-oriented programming design concepts.

3.1 Object-Oriented Software Engineering

3.1.1 Paradigms

The basic assumption of Object-Orientation is that any problem domain can be described in terms of things or entities, which have behavioral characteristics that represent what an entity "does", and structural characteristics that represent what an entity "is" and how it relates to other entities. According to this view, entities with common characteristics can be grouped into classes. The ancient Greek's Theory of Forms, which shares many ideas with Object-Orientation, states that arranging entities into classes (or Forms) is an important way of achieving well-founded knowledge of the world. In other words, "to know the Form of a thing is to understand the nature of that thing". The Theory of Forms suggests that Object Orientation is a natural and intuitive way of analyzing and modeling a problem domain.

Object-Orientation is close to our own natural perception of the real world [17]. The contemporary view of software development takes an object-oriented perspective. In this approach, the main building block of all software systems is the object or class.

Simply put, an object is a thing, generally drawn from the vocabulary of the problem space or the solution space; a class is a description of a set of common objects. Every object has identity (name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behavior (do things to the object, and it can do things to other objects, as well). [2, 17]

3.1.2 The Four Principles of Object-Orientation

Object-Orientation is founded on the following principles:
Abstraction is the formulation of models by focusing on similarities and differences among a set of entities to extract relevant common characteristics, ignoring those aspects that are not relevant to the current purpose. The main goal of abstraction is managing complexity.

Encapsulation (often referred to as *Information Hiding*) facilitates abstraction, by hiding the details of a design decision in a packaged model element. An entity exposes what it is through a specification (or interface), and describes how it is realized by means of an internal implementation. Encapsulation keeps related content together, with the goal of reducing the cost of change.

Inheritance is a mechanism for expressing similarity among entity classes. It allows relating, reusing, and extending representations. The goal of inheritance is to reduce duplication and to prevent inconsistencies.

Polymorphism means that different model elements can have the same specification, but different implementations. This means that the same message can

trigger different operations, depending on the class of the target entity. Polymorphism allows extending existing models with new elements, without having to change the elements that are already in the model.

These principles aim at capturing the world's complexity into maintainable models. The paradigms of encapsulation and polymorphism reduce the cost of change, and – together with abstraction and inheritance – support the management of complexity. The following subsection will introduce the concepts that implement these four principles in the domain of Software Engineering. [17]

3.1.3 Objects, Components, Patterns, Architectures, Frameworks

Object-Orientation regards structural and behavioral characteristics of entities as complete units. For that purpose, object-oriented models are centered around *objects*, which represent (abstractions of) items, persons, or notions. An object describes its structural characteristics by means of *attributes* and *associations* (or *relationships*), and exposes its behavioral characteristics through *operations* (or *services*). Objects communicate with each other by passing *messages*, which cause the recipient to perform an operation, and to return a result to the sender. All objects are grouped into *classes*, which are arranged in an inheritance hierarchy.

Related to classes, the concept of components is fundamental to modern object oriented systems. A *component* is a reusable building block which can be (visually) plugged together with other components. For that purpose, a component exposes a list of *properties* and *services* that other components can link to. Although components are often implemented by a single class, they might also encompass multiple classes. The most widely used libraries of components contain graphical user interface elements like buttons, labels, and lists.

Besides the low-level modeling elements like objects and components, object-oriented methodologies also provide mechanisms to describe larger structures and best-practices. The so-called *Design Patterns* document recurring solutions to

common problems. Patterns have a context in which they apply and must balance a set of opposing consequences or *forces*. Patterns capture modeling experience from which others may learn, and provide a vocabulary (or *Pattern Language*) which allows communicating and discussing design decisions on a high level of abstraction.
Related to architectures is the notion of frameworks. A framework is a collection of several components with predefined co-operations between them. Frameworks allow reusing not only code but architectural design and therefore playing an important role in rapid software development. [17]

3.2 Object-Oriented Dependency

In OO systems, there are important dependences among different objects, such as packages, classes, methods and other code unit in the systems. These dependencies include relationships, contracts and collaborations among objects. These dependencies are determined by different building mechanisms of object-oriented systems: reuse, delegation, data encapsulation, dynamic binding, inheritance and polymorphism. Although these mechanisms represent advantages of object-oriented systems, the maintainers of object-oriented systems must cope with several problems. The most important is that not all these dependencies are explicit in the system.

In any OO system it is widely accepted as good practice to keep the number of interconnections and dependencies between objects to a minimum. The more connections there are then the harder it is to make changes to part of a system without affecting other parts of it. [10]

According to [10] there are two main kinds of dependency within an object-oriented program:

Functional dependencies refer to procedures or sections of code that are called from other sections of code. *Variable dependencies* occur when the attributes contained within objects affect the behavior of the code.

Dependencies also have scope, they may be embedded within a function or method, they may be local to a particular object or they may have remote dependencies that extend across modular boundaries. [10] and [38] gives the definition and two main types, respectively explicit dependency and implicit dependency.

From OO perspective dependency is defined as an object A depends upon another object B, if it is possible that a change to B implies that A is affected or also needs to be changed.

Explicit Dependency: A dependency between two or more objects is explicit when it is precisely and clearly expressed without ambiguity in the source code, i.e., the definition of a direct subclass.

Implicit Dependency: A dependency between two or more objects is implicit when it can be recognized but is not expressed directly in the source code, i.e., the chain of super-classes of a new defined class.

Generally speaking, the dependencies among objects are not all explicit in OO systems. The existence of these implicit dependencies is followed by undesirable characteristics, such as a poorly structured source code, missing or incomplete design specifications, non-existing or out of date documentation, and a high level of redundancy or extremely complex modules. Discovering these dependencies is important if any change is performed in the code.

Furthermore, these meaningful dependencies are determined by different building mechanisms, such as reuse, delegation, data encapsulation, dynamic binding, inheritance and polymorphism. In addition to section 3.1.2

• Class inheritance is the mechanism to define a new class in terms of one or more parent classes. It means that the behavior and data associated with child classes are always an extension of the properties associated with parent classes. A subclass has all the properties of the parent class and others as well. Example: The definition of a class in terms of one or more super-classes is an explicit dependency meanwhile all the chain of super-classes and inherited behavior and state of a class is an implicit dependency.

• Delegation is the mechanism that lets an object delegate to another object whatever behavior the first cannot handle. Example: The delegation of behavior in a method is an explicit dependency meanwhile all the chain of delegates is an implicit dependency.

• Dynamic binding is the mechanism to select lately the method until execution time. It has two main aspects: to determine the object (and the type) and to look up in the chain of super-class for the method. Example: The method lookup made by a chain of super classes is an implicit dependency.

• Data encapsulation is the mechanism to hide the implementation details of classes from users. Users can only invoke the visible methods of classes without knowing how they are implemented.

• Polymorphism describes a variable which refers to a class only known at run time. The polymorphism is shown as: (1) a variable holding a value drawn from a group of types, (2) a name associated with several different method bodies, and (3) a single method with polymorphic variables as parameters. [2, 38]

In [21] author says that a "Good Dependency" is a dependency upon something that is very stable. The more stable the target of the dependency, the more "Good" the dependency is. By the same token a "Bad Dependency" is a dependency upon something that is instable. The more instable the target of the dependency is, the more "Bad" the dependency is.

3.2.1 Software Component Ontology

The author of [38] introduces, Software Component Ontology (SCO) which includes components that summarized from the code comprehension point of view. SCO aims to describe the design of software, to represent software design patterns and related concepts using the concepts developed in that ontology, and to provide a flexible framework for software engineers.

SCO mainly specifies the code structure from the implementation point of view. SCO describes relationships between object-oriented software components (programs that contain namespaces, which contain classes, abstract classes and interfaces, which contain methods and method signatures). Relationships captured include, for example, that an object-oriented class may implement an interface, extend a super class, contain methods, and have membership in a namespace.

Software component ontology is defined to represent the programming elements and their relationships in the source code. Software components normally consist of namespace, packages, modules, classes and so on. Some relation among software components are summarized in Table 3-1.

Table 3-1 Types and Relations Defined in SCO

Relation	Description
is-a	Give the type of an artefact as one of: namespace, class, interface, method, or variable.
definedIn	SourceObject A is defined in SourceObject B.
hasSuper	Class A has super Class B.
name	Relates an artefact to its unqualified name, for example the namespace eHL.Util has name Util.
hasSub	Class A has sub Class B.
scope	Indicates the scope with which the artefact is defined, i.e. public, private or protected.
call	Method A calls Method B.
indirectCall	Transitive relation of method call.

Table 3-1 (Cont'd)

access	Method A read/write Variable B.
In	Indicates the entity within which the artefact is defined. For a method this is a class or interface, for a class or interface it is a namespace, class or interface.
has-param	Indicates that a method has an argument of a particular type.
Read	Method A read Variable B.
write	Method A write Variable B.
extends	Indicates that a class extends another class.
implements	Indicates that a class or interface extends an interface.
hasType	Variable A has Type B.
typeOf	Variable A is of Type B.
kind	Classes and Interfaces are marked as either inner or not-inner.
create	Class A creates instance of Class B.

CHAPTER IV

DEPENDENCIES IN UML

A common practice among software engineers is the use of diagrams for building their system models. Graphical notations become useful for interacting with the end users.

Models are used to describe business processes, to understand the current state of the business, and to model new processes that do not exist but plan to create in the future. Models help to depict business processes and their relationship with other processes.

In this chapter, the benefits of modeling will be listed and the role of Unified Modeling Language (UML) in conceptual design will be described. From the perspective of dependency analysis UML way of dependency representation will be provided. There are a variety of ways that dependency information is represented in UML. UML semantics of relationships with the representations of dependency types will be passed through. Then major advantages and drawbacks will be given in section 4.2.3.

4.1 Object-Oriented Dependency

There are many elements that contribute to a successful software organization; one common thread is the use of modeling. Modeling has been a cornerstone in many traditional software development methodologies for decades. The use of object-oriented modeling in analysis and started to become popular in the late 1980s, producing a large number of different languages and approaches.

UML has taken a leading position in this area, partly through the standardization of the language within the Object Management Group (OMG).

Modeling is a proven and well-accepted engineering technique. Architectural models of houses and high rises to help their users are built to visualize the final product. Even mathematical models are built in order to analyze the effects of winds or earthquakes on the buildings. *A model is a simplification of reality.*

A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction. Every system may be described from different aspects using different models, and each model is therefore a semantically closed abstraction of the system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system. There is one fundamental reason. Models are built so that systems can be better understood.

Through modeling, these four aims are achieved.

1. Models help to visualize a system as it is or as want it to be.
2. Models permit to specify the structure or behavior of a system.
3. Models give a template that guides in constructing a system.
4. Models document the decisions.

Models are built for complex systems because such a system cannot be comprehended in its entirety. There are limits to the human ability to understand complexity. Through modeling, the problem is narrowed by focusing on only one aspect at a time. This is essentially the approach of "divide-and-conquer" that Edsger Dijkstra spoke of years ago: Attack a hard problem by dividing it into a series of smaller problems that can be solved. Furthermore, through modeling, the human intellect is amplified. A model properly chosen can enable the modeler to work at higher levels of abstraction. [2, 25]

4.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such systems. It is intended for use with all development methods, lifecycle stages, application domains, and media. The modeling language is intended to unify past experience about modeling techniques and to incorporate current software best practices into a standard approach. UML includes semantic concepts, notation, and guidelines. It has static, dynamic, environmental, and organizational parts. It is intended to be supported by interactive visual modeling tools that have code generators and report writers. The UML specification does not define a standard process but is intended to be useful with an iterative development process. It is intended to support most existing object-oriented development processes. [25]

4.2.1 Concepts of the UML 2.0

General UML concepts are used regularly, so they will be broadly described here, before entering into more specific aspects of UML 2.0. These include the concept of Class, object and relationships. [3]
Class is a category or group of items that have the same attributes (properties) and the same behaviors (operations). Classes that do not have parents are called base classes. Base classes can be both abstract and concrete meta-classes from the UML meta-model [14].

A simple example of a Class is a Student. The class “Student” has some attributes (e.g.: IDStudent, name, surname, dateOfBirth) and behaviors (e.g.: IntroduceStudent(), UpdateStudent ( ), RemoveStudent( )). Figure 4-1 shows the graphical example of the class Student and its instance - the object Marie.

Thus, if a particular Student is indicated, for example, Marie, and its respective attributes (010101-P111, Marie, Lee, 01/01/01), this is called object. *Object* is an instance of a class, i.e., a specific thing that has specific values of the classes attributes [3].

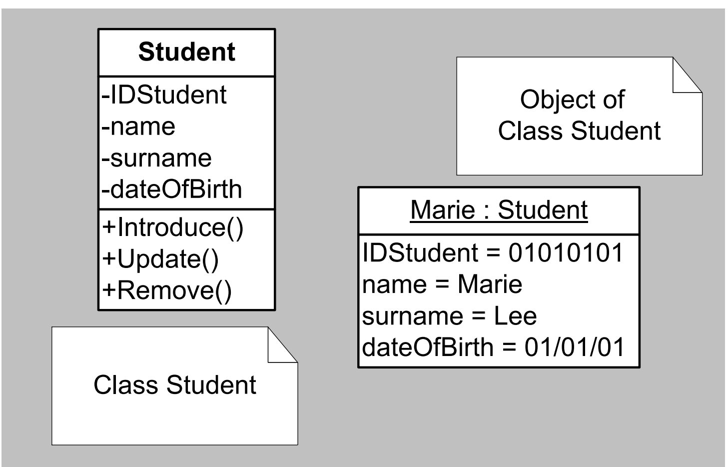


Figure 4-1 Example of Class and Object

There is a special type of class denominated *abstract class. Abstract class* is a class that provides an operation signature, but no implementation [2]. These classes are useful for identifying common functionality across several types of objects. For example, an abstract class denominated Movable and one operation defined as move(). Because the base class Movable does not have an implementation for move(), the class is said to be abstract [2].

Classes are related to each other through *relationships*. Each UML relationship represents a different type of connection between classes. There are different types of relationships. Here the most used in metamodelling language have been listed below. The most common relationships and their characteristics are provided in Table 4-1. Table 4-1 shows the graphic representation and how to read the relationships included in "<<>>" annotations [3].

Table 4-1 UML Relationship Types [13, 14]

Type of Relationship	Graphical Representation	Relationship Characteristic
Dependency	A «uses» B A <<uses a >>B	Weakest relationship in which one class uses the knowledge of the other class.
Association	A B A <<has a>> B	Indicate that one class keeps a relationship with another over an extended period of time.
Aggregation	A «uses» B	This is a stronger version of association. It implies ownership and may imply a relationship between lifelines.
Composition	A B 1 * B << is part of >> A	It represents a very strong relationship, to the point of containment, i.e., it is used to capture the whole-part relationship
Generalization	A B B <<is a>> subclass of A	It represents an inheritance relationship, in which the subclass inherits all features of the parent and may add its own.

4.2.2 Dependencies

A *dependency* indicates a semantic *relationship* between two or more model elements. It relates the model elements themselves and does not require a set of instances for its meaning. It indicates a situation in which a change to the supplier

element may require a change to or indicate a change in meaning of the client element in the dependency.

The *association* and *generalization* relationships are dependencies by this definition, but they have specific semantics with important consequences. Therefore, they have their own names and detailed semantics. The word dependency is normally used for all the other relationships that don't fit the sharper categories.

Table 4 - 2 lists the kinds of dependency found in the UML base model. A trace is a conceptual connection among elements in different models, often models at different stages of development. It lacks detailed semantics. It is typically used to trace system requirements across models and to keep track of changes made to models that may affect other models.

A *refinement* is a relationship between two versions of a concept at different stages of development or at different levels of abstraction. The two concepts are not meant to coexist in the final detailed model. One of them is usually a less finished version of the other.

In principle, there is a mapping from the less finished concept to the more finished concept. This does not mean that translation is automatic. Usually, the more detailed concept contains design decisions that have been made by the designer, decisions that might be made in many ways. In principle, changes to one model could be validated against the other, with deviations flagged. In practice, tools cannot do all this today, although some simpler mappings can be enforced. Therefore a refinement is mostly a reminder to the modeler that multiple models are related in a predictable way.

A *derivation* dependency indicates that one element can be computed from another element (but the derived element may be explicitly included in the system to avoid a

costly re-computation). Derivation, realization, refinement, and trace are abstraction dependencies - they relate two versions of the same underlying thing.

A *usage* dependency is a statement that the behavior or implementation of one element affects the behavior or implementation of another element. Frequently, this comes from implementation concerns, such as compiler requirements that the definition of one class is needed to compile another class. Most usage dependencies can be derived from the code and do not need to be explicitly declared, unless they are part of a top-down design style that constrains the organization of the system (for example, by using predefined components and libraries). The specific kind of usage dependency can be specified, but this is often omitted because the purpose of the relationship is to highlight the dependency. The exact details can often be obtained from the implementation code. Stereotypes of usage include *call* and *instantiation*. The *call* dependency indicates that a method on one class calls an operation on another class; instantiation indicates that a method on one class creates an instance of another class.

Table 4-2 Kinds of Dependencies

Dependency	**Function**	**Keyword**
access	Permission for a package to access the contents of another package	access
binding	Assignment of values to the parameters of a template to generate a new model element	bind
call	Statement that a method of one class calls an operation of another class	call
derivation	Statement that one instance can be computed from another instance	derive
friend	Permission for an element to access the contents of another element regardless of visibility	friend
import	Permission for a package to access the contents of another package and add aliases of their names to the importer's namespace	import
instantiation	Statement that a method of one class creates instances of another class	instantiate
parameter	Relationship between an operation and its parameters	parameter

Table 4-2 (Cont'd)

realization	Mapping between a specification and an implementation of it	realize
refinement	Statement that a mapping exists between elements at two different semantic levels	refine
send	Relationship between the sender of a signal and the receiver of the signal	send
trace	Statement that some connection exists between elements in different models, but less precise than a mapping	trace
usage	Statement that one element requires the presence of another element for its correct functioning (includes call, instantiation, parameter, send, but open to other kinds)	use

Several varieties of usage dependency grant permission for elements to access other elements. The *access* dependency permits one package to see the contents of another package. The *import* dependency goes further and adds the names of the target package contents to the namespace of the importing package. The *friend* dependency is an access dependency that permits the client to see even the private contents of the supplier.

A *binding* is the assignment of values to the parameters of a template. It is a highly structured relationship with precise semantics obtained by substituting the arguments for the parameters in a copy of the template.

Usage and binding dependencies involve strong semantics among elements at the same semantic level. They must connect elements in the same level of model (both analysis and both design, and at the same level of abstraction). Trace and refinement dependencies are vaguer and can connect elements from different models or levels of abstraction.

The instance of relationship (a metarelationship, not strictly a dependency) indicates that one element (such as an object) is an instance of another element (such as a

class). A dependency is drawn as a dashed arrow from the client to the supplier, with a stereotype keyword to distinguish its kind, as shown in Figure 4.2. [20]

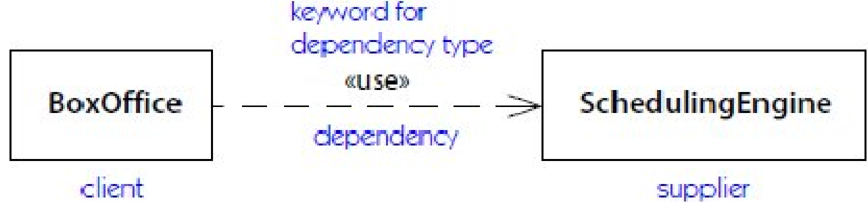


Figure 4-2 Dependencies

4.2.3 UML Characteristics With Respect To Dependency

In [6] authors describe the dependency considering the basic UML *dependency* construct (denoted by a simple dashed arrow) as a "using" relationship indicating directional links between elements (often classes). The meaning of this construct is that element A uses element B, so that a change in class B may require or cause a change to element B. This constitutes a dependency by our definition as well; however, formal semantics relating to the dependency are still lacking. The dependency construct may be considered to represent a general "functional dependency" but it is not clear what a user of a UML diagram is supposed to do with it. What constraints must therefore be observed? What additional meaning is implied by the dependency? For simple traceability analysis, dependency arrows are helpful in identifying where changes may be implied, but there is no further guidance to a UML modeler as to what kind of changes the dependency implies, so that the modeler needs to label the dependency arrows in a UML model with additional information concerning the dependency type.

UML's extensibility mechanisms can extend its notation; however, they do not in general extend any semantics. There are other dependencies implicit in various other UML diagrams. In fact, dependencies in UML are spread out over all of the UML diagrams, and generally implicit. Each different kind of implicit dependency has its own characteristics (e.g., presence or absence of transitivity) and set of rules.

This is not just a matter of inconvenience; it also means that for each construct there exists a variety of differing interpretations, each one of which affects what the dependency may mean. This makes sharing knowledge about the dependencies all the more problematic.

Space does not permit to examine each construct in UML, but it is claimed that most (if not all) of them do not possess inherent semantics and therefore are lacking in sufficient power to represent dependencies for formal modeling. Further it is claimed that there is no uniform representation for dependency in UML, nor is there a way to specify dependency semantics beyond UML's original semantics.
UML has the advantage of using multiple diagram types to model each aspect of a system. This permits each diagram to be tuned to each aspect of a problem. The difficulties are that a user must learn and master multiple types and their appropriateness and syntax. Further, the relationship of one type of diagram to another is defined only in informal senses, with the idea that each representation is more or less orthogonal to the others. This makes it difficult to perform automatic comparisons, inferences and queries.

UML's characteristics with respect to dependency can be summarized as follows. On the plus side, it possesses good generality in that its diagrams cover a wide range of system characteristics and features. Its drawbacks are:

- no underlying built-in semantics for dependencies,
- several different ways to represent dependencies
- not overall formal semantic description that allows dependencies to fit formally into a complete description.

CHAPTER V

DESIGN STRUCTURE MATRICES

A Dependency Structure Matrix (DSM) is a means of representing the interactions between elements. In this chapter a thorough introduction to (DSMs) will be presented. The main topics cover the origins and basic characteristics of DSMs. The DSM literature and basic concepts for DSM will be built and DSM types and main roles will be investigated.

5.1 Basic Concepts

The Dependency Structure Matrix (DSM) is a complexity management technique that has proved to be valuable for managing, designing, modeling, analyzing and optimizing technical systems, complex organizations, sizeable engineering projects, densely networked processes and large market structures [27].

A Design Structure Matrix is a tool that can be used to model complex systems. A DSM is a compact form for representing the dependencies in a system. The matrix puts the units (parameters, subsystems, activities, tasks) and indicates the dependencies among those.

Depending on the context, the dependency patterns can represent different aspects of the system or project.

DSMs where first conceived by Donald Steward at General Electric in the late 1960s, but it took until 1981 before his work was published [7]. In his work Steward proposed a novel method, called the Design Structure System, to manage the complexity of large systems or engineering projects using DSMs.

Eppinger et al. [8] extended this model to capture more deeply the correspondence between design structure and task structure in product development.

Despite being conceived as a project management tool, a DSM is an analysis and design instrument that lends itself to a multitude of other applications across a wide range of domains and disciplines. The interest for DSMs from computer science and software development in particular is growing.

The DSM community maintains a portal website [27] which lists all publications and all knowledge on the subject. The community meets at the annual International DSM Conference [28] that has been sponsored by large corporations such as Boeing and the BMW Group.

5.2 Dependencies

Consider a system that is composed of two elements - sub-systems (or activities/phases): element "A" and element "B". A graph may be developed to represent this system pictorially. The graph is constructed by allowing a vertex/node on the graph to represent a system element and an edge joining two nodes to represent the relationship between two system elements. The directionality of influence from one element to another is captured by an arrow instead of a simple link. The resultant graph is called a directed graph or simply a digraph. There are three basic building blocks for describing the relationship amongst system elements: parallel (or concurrent), sequential (or dependent) and coupled (or interdependent) [37].

Sequential relationships imply that activities must be carried out in sequence, that the output of one activity is required for the next. A must come before B, or B is dependent of A, see Figure 5-1.

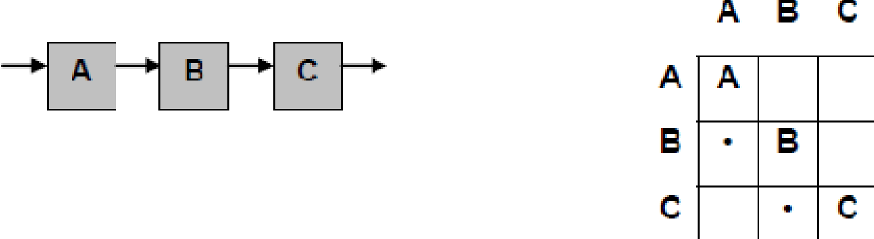


Figure 5-1 Sequential relationship (Dependent)

If activities are not dependent of each other they are independent and can be carried out parallel to each other. In a matrix there would not be any marks to indicate dependencies. See Figure 5-2.

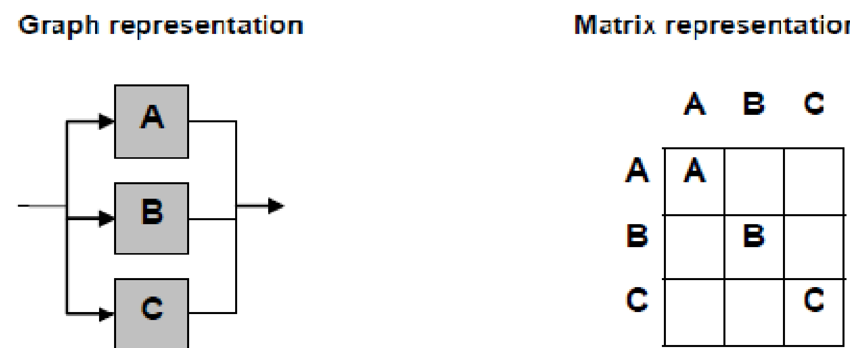
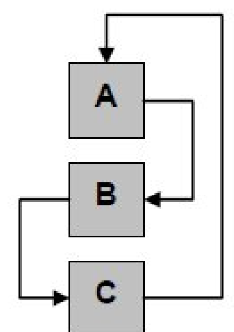


Figure 5-2 Parallel relationships (Concurrent)

The complexity in a process occurs when activities become interdependent. C is dependent on the output of B and B is dependent on the output of A but A requires information from C. This is a clear example of iteration from C to A.See Figure 5-3.

Graph representation



Matrix representation

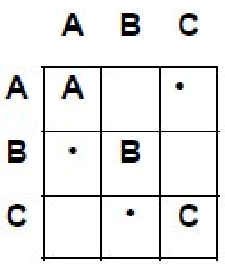


Figure 5-3 Coupled relationships (Interdependent)

In the parallel configuration, the system elements do not interact with each other. Understanding the behavior of the individual elements allows to completely understanding the behavior of the system. If the system is a project, then system elements would be project tasks to be performed. As such, activity B is said to be independent of activity A and no information exchange is required between the two activities. In the sequential configuration, one element influences the behavior or decision of another element is a unidirectional fashion. That is, the design parameters of system element B are selected based on the system element A design parameters. Again, in terms of project tasks, task A has to be performed first before task B can start. Finally, in the coupled system, the flow of influence or information is intertwined: element A influences B and element B influences A. This would occur if parameter A could not be determined (with certainty) without first knowing parameter B and B could not be determined without knowing A. This cyclic dependency is called "Circuits" or "Information Cycles". [37]

5.3 Types of DSMs

Tyson Browning [32, 33] distinguishes four different types of DSM applications, based on the kind of data that is represented. He also introduced two main categories static and time-based. Figure 5-4 shows taxonomy of the categories and types of DSMs according to Browning.

In static DSMs the parameters represent the elements of a system that exist simultaneously, such as components of product architecture or groups of people in an organization. In time-based DSMs the parameters represent activities or processes, and their ordering in the matrix indicates a flow through time, or in other words, the chronological order in which they are to be carried out.

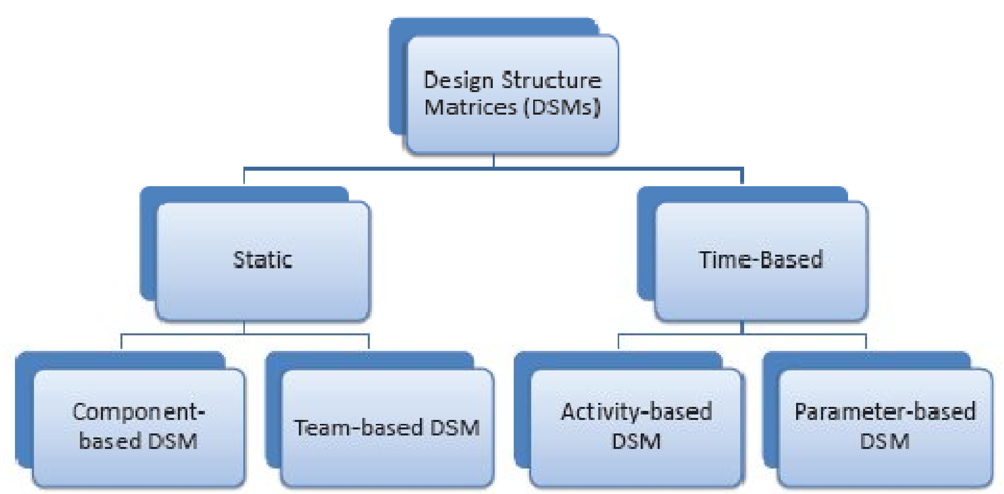


Figure 5-4 A taxonomy of DSM types according to Browning[32, 33]

DSMs are often processed with metrics or algorithms that analyze and/or restructure the representation of a project or a system. In [33], Browning discusses each of the four types of DSMs and their accompanying analysis methods using industrial examples.

Component-based or Architecture DSMs are useful for modeling system architectures, involving relationships and interactions among components or subsystems and for facilitating appropriate decomposition strategies. Component-based DSMs can be combined with clustering algorithms, which localize dependencies by defining subsets of parameters with minimal external dependencies. Such subsets are called clusters or chunks, but largely correspond to the definition of modules.

Team-based, People-based or Organization DSMs are used to design integrated organization structures based on (groups of) people and their interactions. These DSMs can also be combined with clustering algorithms. Activity-based, Task-based or Schedule DSMs are suited to model the information flow and other dependencies among processes and their constituent activities. (Re)Sequencing methods can be used to optimize the chronological order of the activities in activity-based DSMs [33, 37]. An example of such an algorithm is partitioning, which transforms the matrix into a nearly lower triangular form in order to minimize feedbacks (activities that depend on activities in the future).

Finally, Parameter-based or Low-Level Schedule DSMs are aimed at modeling and integrating low-level design decisions and processes based on physical design parameter relationships. A noteworthy example can be found in the work of Black et al. who applied parameter-based DSMs to an automobile brake system design [29].

Table 5-1 Four different types of data that can be represented in a DSM [33]

DSM Data Types	**Representation**	**Application**	**Analysis Method**
Task-based	Task/Activity input/output relationships	Project scheduling, activity sequencing, cycle time reduction	Partitioning, Tearing, Banding, Simulation and Eigenvalue Analysis
Parameter-based	Parameter decision points and necessary precedents	Low level activity sequencing and process construction	Partitioning, Tearing, Banding, Simulation and Eigenvalue Analysis
Team-based	Multi-team interface characteristics	Organizational design, interface management, team integration	Clustering
Component-based	Multi-component relationships	System architecting, engineering and design	Clustering

5.4 Roles of DSMs

According to [27], a DSM is both a project management tool and system analysis tool. In the role of a project management tool a DSM is primarily used to diagram information flows in complex projects. As a system analysis tool DSMs are used to analyze processes and architectures of products or organizations. However, there is no clear boundary between both roles, as specific applications of DSMs can combine them.

In these applications, DSMs represent the constituent stages, tasks or activities of an engineering project, along with the corresponding dependencies. The dependencies define the input which is required to start a certain activity and the generated output that needs to feed into other activities. Both input and output represent pieces of information. Hence, the pattern of dependencies in the DSM explicitly describes the exchange of information that is vital to the project [27].

Traditional project management tools, such as PERT charts, Gantt charts and Critical Path methods (CPM) were created to model and manage sequential and parallel processes consisting of discrete tasks that make up large construction projects. They capture work flow, often using pre and post conditions (e.g.: Which tasks must be completed before task X can start?"), but do not to track the flow of information (e.g.: Which pieces of information are needed before task X can succeed?").

Compared to conventional project management tools, DSMs focus on representing information flows rather than work flows. Therefore, the DSM method, which is essentially an information exchange model, enables managers and product development planners to deal with the complex relationships in large engineering projects [37].

5.4.1 Project Management Tool

The Design Structure Matrix has its origins in project management and is still used in that context by large corporations such as General Motors, Boeing, Airbus and Intel. Project management applications of DSMs continue to receive interest from the research community as well [37].

5.4.2 System Analysis Tool

DSMs can also be applied as a tool to analyze complex systems. Analyzed system architectures can be for both tangible (e.g.: material products) and intangible things (e.g.: projects or organizations). The compact and clear representation DSMs provide facilitates the capturing and understanding of interactions, interdependencies and interfaces between the elements of the system, such as subsystems or modules. Moreover, the diagrams can highlight key processes and enable engineers to discover previously unknown patterns in architectures. The diagrams can also show where staff members fit in the larger project or organization they are part of [37].

CHAPTER VI

DSM-BASED VISUALISATION

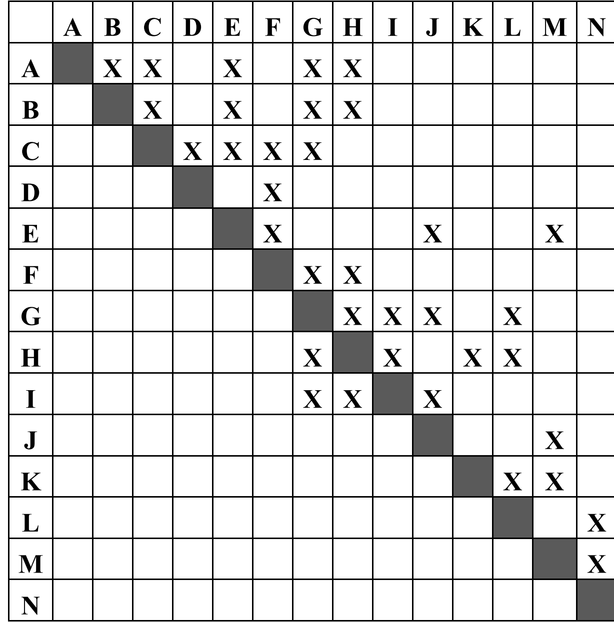
In this chapter the basic topics from a major resource the DSM website (www.DSMweb.org) that includes lots of examples, cases, tutorial, references, and computer macros to perform partitioning, tearing, clustering, and simulation will be presented. It also contains links to other DSM researchers and research institutions.

And also, DSM visualization topics will be elaborated and some examples will be given for the simplicity of understanding the DSM concept. In the last section commercial and non commercial products are presented which uses DSM.

6.1 Reading, Partitioning, Clustering

6.1.1 Reading a DSM

The figure below shows a DSM model for the 14 major tasks. The X marks indicate the existence and direction of information flow (or a dependency in a general sense) from one activity in the project (represented by the overall matrix) to another. Reading across a row reveals the output of a task by an X mark placed at the intersection of that row with the column that bears the name of the receiving task. Reading across a column reveals the input information flows to that activity to other activities by placing an X in a similar manner described above. For example, consider activity C in the above matrix. Activity C relies on information from activities A and B, delivers information to activities D, E, F and G. The marks (above the diagonal) thus represent the forward flow of information.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A		X	X		X		X	X						
B			X		X		X	X						
C				X	X	X	X							
D						X								
E						X				X			X	
F							X	X						
G								X	X	X		X		
H							X		X		X	X		
I							X	X		X				
J													X	
K												X	X	
L														X
M														X
N														

Figure 6-1 Sample DSM

The marks (below the diagonal) are of special significance. Such a mark reveals a feedback from a later (i.e. downstream) activity to an earlier (i.e. upstream) one. This means that the earlier activity may have to be repeated or reworked in light of the late arrival of new information. Such an iterative process is common in most engineering design and development projects. Design iterations create rework and require extra communication and negotiation which result in a prolonged development process. In order to speed up this iterative design process, the DSM methodology suggests the manipulation of the matrix elements such that iterative behavior is removed from the matrix, or at least minimized (a process called Sequencing, Triangularization, Block Diagonalization, or Partitioning). [27]

6.1.2 Partitioning a DSM

The removal of iterations is referred to as sequencing or partitioning. The matrix is divided in two sections by a diagonal line (each element intersecting itself). Every mark in the upper triangle symbolizes iterations (See Figure 6.2). The DSM rearranges the order in which tasks are carried out to move these marks to the lower triangle (www.dsmweb.org). In Figure 6.2 the left matrix illustrate a fabricated

process with its dependencies. It also gives the order of when the tasks are issued. A is the first task in the process, G is the last. This arrangement of activities has six iterations, the largest spanning from G to B. The right matrix provides the optimal solution obtained through a partitioning algorithm. By rearranging the order of tasks (F-B-D-G-C-A-E) only two minor iterations remain.

	A	**B**	**C**	**D**	**E**	**F**	**G**
A	A		•	•			
B		B					•
C	•	•	C		•		•
D		•		D			
E			•		E	•	
F						F	
G				•		•	G

	F	**B**	**D**	**G**	**C**	**A**	**E**
F	F						
B		B		•			
D		•	D				
G	•		•	G			
C	•	•		•	C	•	
A					•	A	
E	•				•		E

Figure 6-2 Example of Partitioning

Partitioning a DSM aims at finding the optimal order of activities. To do so the first thing is to understand what components the project consists of. After this all of the activities that make up each component are listed. When this is done one can start looking on who does what and what time the activity takes (duration). The time each worker spends on an activity must also be anticipated i.e. the workload. [15]

6.1.3 Clustering a DSM

A different method of analysis is by grouping elements that have the most interaction into modules along the diagonal. These modules consist of both sequential relationships and iterations. Tasks that share the most information should be carried out in close proximity; this method is known as clustering (Figure 6.2). The aim of clustering is significantly different from partitioning, the objective is to increase iterations within the modules but remove any iteration between the

modules. Clustering is useful when dealing with representation of design components or in shaping project development teams. Areas between clusters may also contain dependencies. These areas are referred to as the interface between clusters. Dependencies in the interface share information to at least two clusters and must be carefully coordinated. They are the links between the modules. [4]

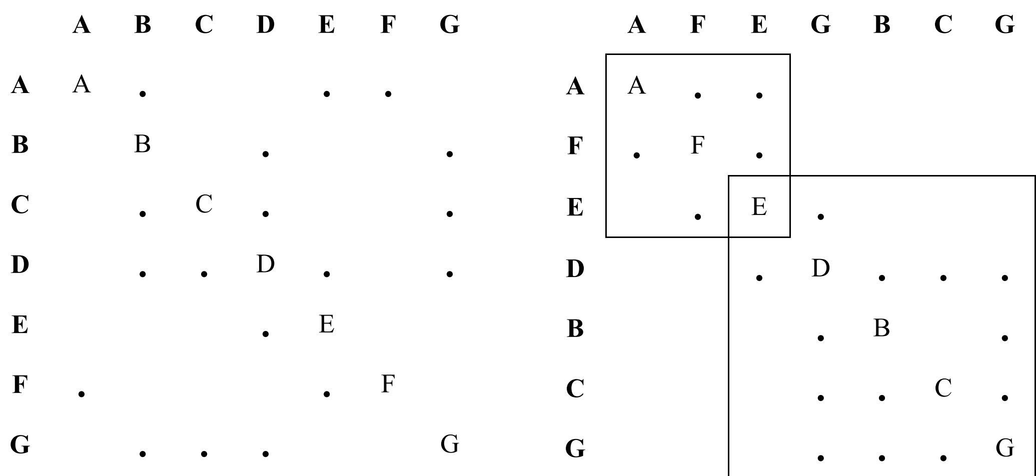


Figure 6-3 Example of Clustering

6.1.4 Numerical DSMs

In binary DSM notation (where the matrix is populated with "ones" & "zeros" or "X" marks & empty cells) a single attribute was used to convey relationships between different system elements--namely, the "existence" attribute which signifies the existence or absence of a dependency between the different elements.

Compared to binary DSMs, Numerical DSMs (NDSM) could contain a multitude of attributes that provide more detailed information on the relationships between the different system elements. An improved description or capture of these relationships provides a better understanding of the system and allows for the development of more complex and practical partitioning algorithms.As an example, consider the case where task B depends on information from task A. However, if this information is predictable or have little impact on task B, then the information

dependency could be eliminated. Binary DSMs lack the richness of such an argument. Possible attributes and measures that can be used:

- Level Numbers: Steward suggested the use of level numbers instead of a simple "X" mark, for certain marks in the binary matrix. Level numbers reflect the order in which the feedback marks should be torn. The mark with the highest level number will be torn first and the matrix is reordered (i.e., partitioned or sequenced) again. This process is repeated until all feedback marks disappear. Level numbers range from 1 to 9 depending on the engineers judgment of where a good estimate, for a missing information piece, can be made.

- Importance Ratings: A simple verbal scale can be constructed to differentiate between different important levels of the "X" marks. As an example, a 3-level scale can be defined as follows:

Table 6-1 Importance Ratings

Numeric Scale	Meaning
1	High Dependency
2	Medium Dependency
3	Low Dependency

Some other attributes depend on the type of DSM used in the representation and analysis of the problem. For example, in an Activity-based DSM, the following measures can be used:

Dependency Strength: This can be a measure between 0 and 1, where 1 represents an extremely strong dependency. The matrix can, now, be partitioned by minimizing the sum of the dependency strengths below the diagonal.

Volume of Information Transferred: An actual measure of the volume of the information exchanged (measured in bits) may be utilized in the DSM. Partitioning of such a DSM would require a minimization of the cumulative volume of the feedback information.

Variability of Information Exchanged: A variability measure can be devised to reflect the uncertainty in the information exchanged between tasks. This measure can be the statistical variance of outputs for that task accumulated from previous executions of the task (or a similar one).

Probability of Repetition: This number reflects the probability of one activity causing rework in another. Feedback relationships represent the probability of having to loop back (i.e. iteration) to earlier (upstream) activities after a downstream activity was performed, while feed-forward relationships can represent the probability of a second-order rework following an iteration.

Impact strength: This can be visualized as the fraction of the original work that has to be repeated should iteration occur. This measure is usually utilized in conjunction with the probability of repetition measure, above, to simulate the effect of iterations on project duration.

6.2 Representing Dependencies Using DSMs

An object-oriented system is composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. Similar objects form classes, which provide the static description of the properties and behaviors that their instances will have. Therefore, extracting, analyzing, and modeling classes/objects and their relationships is of key importance in acquiring in-depth understanding of object-oriented software systems. However, when dealing with complex object-oriented systems, maintainers can easily be overwhelmed by the large number of classes/objects and the high degree of interdependencies among them [36]. A commonly used strategy to address the scalability problem is to partition the set of all classes/objects into coarse-grained container entities and then analyze their interrelationships.

While some modules in object-orientation, such as objects or methods, are apparent as explicit units of code, this is not always true for others, such as packages or

namespaces. Most dependencies in object-orientation are explicitly defined, but often by individual lines of code (e.g.: a method call or the specification of inheritance relations in a class header), which makes them hard to track down. However, a DSM-based visualization can display all modules in the same, explicit, way and can offer a convenient overview of all dependencies, no matter where or how they are defined.

Figure 6-4 shows such a method-class-package DSM series for a hypothetical piece of object-oriented software.

As an example this section uses DSMs for analyzing the dependencies of the three simple packages. Only direct dependencies are included on the method-level to emphasize how higher level DSMs summaries the dependencies on lower levels (i.e.: M-DSM sums method-level dependencies per ordered pair of classes and P-DSM does so per ordered pair of packages).

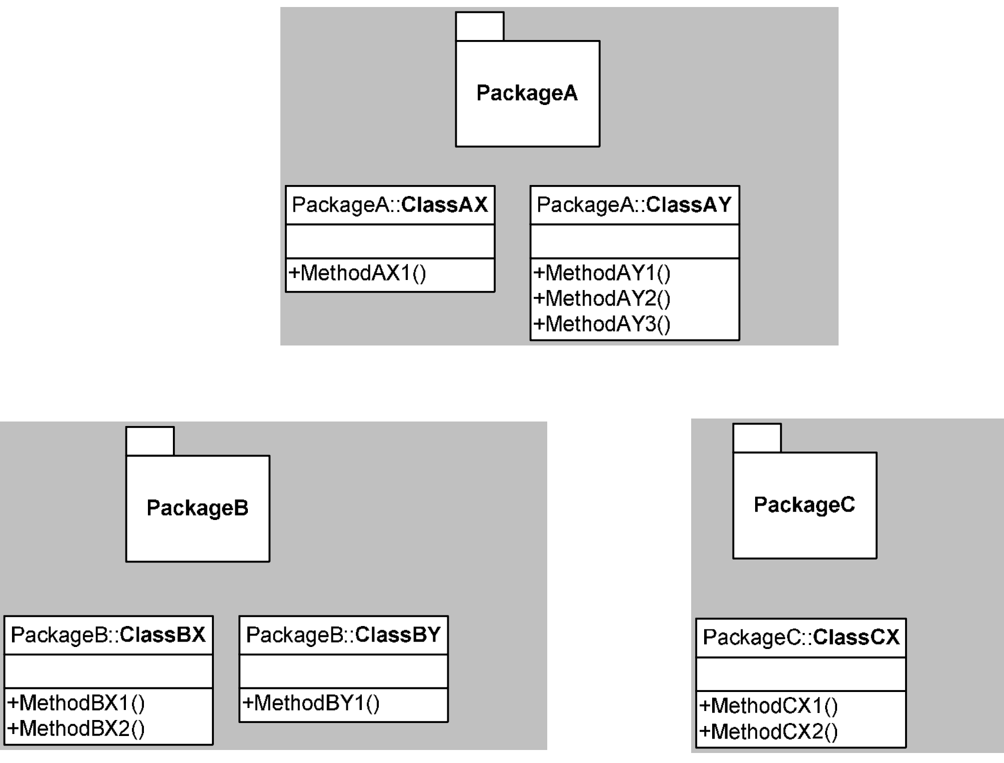


Figure 6-4 Sample Packages

The UML diagram in Figure 6-4 only displays three packages with related classes and methods for the simplicity the dependencies originating from the packages are not shown. Now those dependencies to illustrate the summarizing process in the DSM series shown in Figure 6-5 is used.

Method relationships, flow of message from one to the next is significant. At this stage these are noted on a binary scale of 0 (no dependency exists) or 1 (a dependency exists). All method-level dependencies are caused by method calls are given an individual weight of 1.

Looking at M-DSM, the dependencies originating from Package C are caused by 7 method calls to 5 different methods (one is called 3 times from the same method). Next, in C-DSM the 7 method calls target methods in 4 classes (respectively 3, 1, 2 and 1 times). Finally, in P-DSM the method calls from Package C target methods in classes of both Package A and Package B (respectively 4 and 3 times). [26]

Package Level P-DSM				
		1	2	3
Package A	1	•	2	
Package B	2	6	•	1
Package C	3	4	3	•

Figure 6-5 Sample DSMS Package Level

Class Level C-DSM						
		1	2	3	4	5
Class AX	1	•			1	
Class AY	2	2	•		1	
Class BX	3	1	3	•	1	
Class BY	4		2	1	•	1
Class CX	5	3	1	2	1	•

Figure 6-6 Sample DSMS Class Level

Method Level M-DSM										
		1	2	3	4	5	6	7	8	9
MethodAX1	1	•						1		
MethodAY1	2	1	•		1					
MethodAY2	3		1	•						
MethodAY3	4	1			•			1		
MethodBX1	5	1	1			•				
MethodBX2	6			2			•	1		
MethodBY1	7		1			1		•		1
MethodCX1	8			1	1		1		•	
MethodCX2	9	3				1				•

Figure 6-7 Sample DSMS Method Level

6.3 Related Work

Although the application of DSMs in support of software development is a fairly recent phenomenon, other parties have conducted related research and created similar tools. Three such tools, Lattix LDM and NDepend are discussed, which are the most mature examples, and an experimental program called DeMatrix.[26]

6.4 Lattix LDM

Lattix, Inc. was the first company to release a commercial support tool for software development which applies DSMs as abstract representations of software implementations. The product is called Lattix LDM and is primarily promoted as a tool for analyzing and managing large-scale software development projects. Lattix LDM and its underlying methodology have been demonstrated in talks at conferences and in a number of articles. A trial version have experimented with Lattix LDM [26].

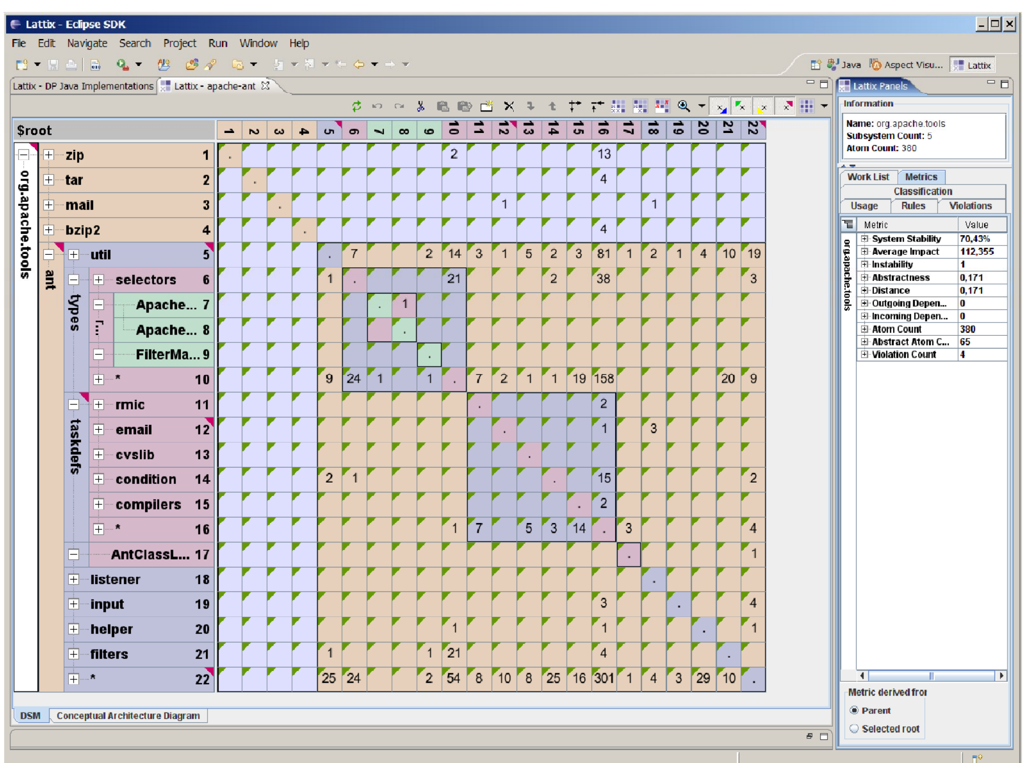


Figure 6-8 Lattix LDM Eclipse plug-in showing a DSM for Apache Ant

Lattix LDM reverse engineers Java, C/C++ and .NET code to DSM diagrams and comes as a stand-alone application (for Windows or Linux) and as a plug-in for the Eclipse development environment . Figure 6-6 shows a screenshot of the Eclipse plug-in displaying a DSM visualization of the Apache Ant source code. [26]

The user interface of Lattix LDM uses a tree-based DSM visualization, which aggregates classes per package level and allows packages to be collapsed and expanded. While this was a source of inspiration for DSMBrowser, Lattix does not offer the same level of detail, as it does not document dependencies below the level of classes.[26]

Lattix computes package-level dependencies by summing class-level dependencies. Class level dependency values (which are referred to as dependency strengths) are numerical and can be configured to be knowledge-based on usage-based. The available documentation lacks a formal explanation of both configurations, but as far as the knowledge based configuration only takes into account which classes know one another, while the usage-based configuration expresses the degree to which classes use each other's functionality. Knowledge-based dependency strengths seem to be limited to a scale from 0 to 2.

The usage-based configuration results in a much wider range of dependency strengths and clearly provides a more detailed approximation of implementation level dependencies. However, due to the lack of method-level dependencies, even Lattix LDM's usage-based configuration fails to provide the same level of detail DSMBrowser offers [26].

Other notable features of Lattix LDM include a number of dependency filtering settings and the definition of design rules to capture and enforce architectural intent. Software architects can create design rules with Lattix LDM to express the nature of dependencies between subsystems or classes. Dependencies that violate such design rules are then highlighted in the visualization. [26]

6.5 NDepend

NDepend, is another commercial software development tool that applies DSM diagrams. It is a dependency management tool that is intended to facilitate controlling the complexity, quality and evolution of source code. NDepend exclusively targets the .NET software development platform and integrates with the Microsoft Visual Studio IDE [26].

The tool analyses source code and compiled .NET assemblies to generate reports and interactive graphical visualizations, based on tree-based DSMs and other diagrams. It also includes over 60 predefined metrics to analyze different aspects of software implementations. Moreover, it provides metaprogramming facilities by means of an SQL-like query language called Code Query Language (CQL), which allows users to write queries against the code structure of .NET applications and which can be used to write custom metrics. [26]

6.6 DeMatrix

DeMatrix is a tool created by Sushil Bajracharya et al. at the University of California, Irvine. It was developed in connection with a larger research project that aims to create an infrastructure, named Sourcerer, for large-scale analysis of open source code repositories. DeMatrix is a front-end for Sourcerer that visualizes software using DSMs.

Currently DeMatrix is only available as a Java applet embedded on a demonstration webpage http://mine7.ics.uci.edu/repo2/dsm.html. The applet displays DSM visualizations of Java source code of various open source projects hosted at SourceForge http://sourceforge.net. The screenshot in Figure 6-7, on the previous page, shows the DeMatrix applet visualizing the code of the JAPAN project.

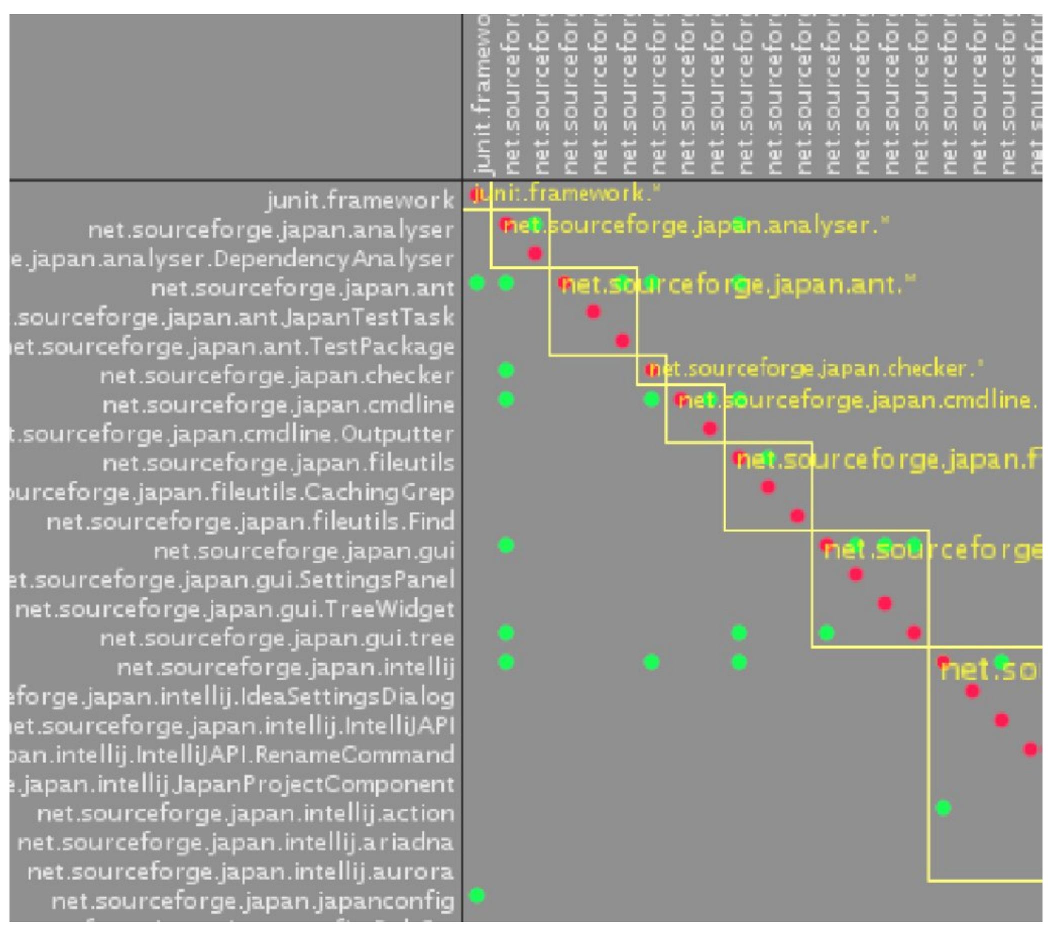


Figure 6-9 DeMatrix applet displaying a DSM for the source code of JAPAN

DeMatrix provides a basic, fairly static visualization based on (binary) DSMs and no analysis features. The design parameters, which are confronted in the DSM, correspond to Java classes. Modules, corresponding to packages, are indicated using yellow bordered boxes on the diagonal. The DSM is not tree-based so packages cannot be collapsed or expanded. The module level can be shifted to correspond to a higher or a lower package level, which respectively results in larger or smaller module boxes but does not otherwise alter the visualization.

CHAPTER VII

DSM INTERFACE IMPLEMENTATION

In this chapter a tool to support the visualization of DSMs will be implemented and optimization of system dependencies by using the partitioning algorithm. First some well known algorithms will be introduced. Then the sample application developed using C#.Net will be shown.

7.1 Algorithms

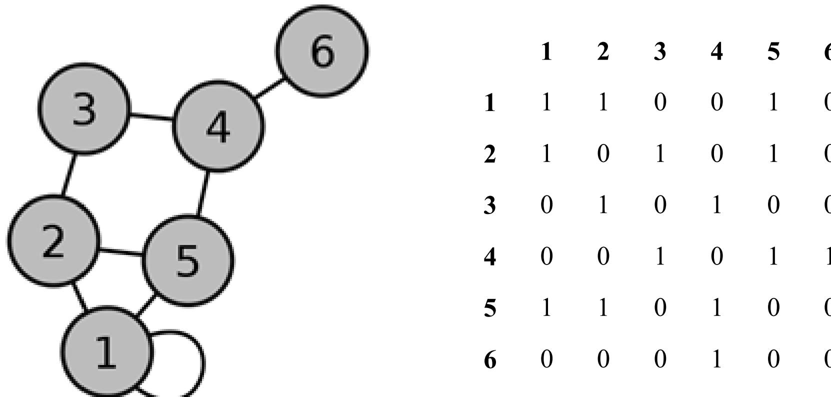
This section will present some graph representations that are commonly used in computer science. Some methods for manipulating these data structures will also be discussed. [22]

Adjacency Matrix: The digraph model of the dependency is a useful mathematical construct for depicting the relationships between the objects, components or methods, but is of little use for computational purposes. Hence, the graph must be modeled by using a data structure that is simple to operate on by means of programming. [22]

In mathematics and computer science, an adjacency matrix is a means of representing which vertices of a graph are adjacent to which other vertices. Specifically, the adjacency matrix of a finite graph G on n vertices is the $n \times n$ matrix where the non-diagonal entry a_{ij} is the number of edges from vertex to vertex

j, and the diagonal entry a_{ii}, depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself. Undirected graphs often use the former convention of counting loops twice, whereas directed graphs typically use the latter convention.

There exists a unique adjacency matrix for each graph (up to permuting rows and columns), and it is not the adjacency matrix of any other graph. In the special case of a finite simple graph, the adjacency matrix is a (0, 1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric. [11] (Figure 7-1).



	1	**2**	**3**	**4**	**5**	**6**
1	1	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Figure 7-1 An example directed graph and its corresponding adjacency matrix

The parameter dependency graph that will be represented by an adjacency matrix imposes unique restrictions on it, namely:

In case of a digraph representing a dependency structure, a self-loop on a node would indicate that the corresponding item depend on itself an impossible situation. Thus, the adjacency matrix of the digraph must have a zero trace vector (i.e. zero elements on the main diagonal).

Between each two connected knowledge items a and b, there must be at most one directed edge from a to b, and at most one from b to a. For each two elements a, b in

the graph, if there's a relationship between a and b such that a depends on b, then the corresponding entry in the adjacency matrix M[a; b] = 1.Otherwise, M[a; b] = 0.

Reachability Matrix: While the adjacency matrix represents only the direct connections between the nodes, it cannot show the indirect dependencies between the items. The *reachability matrix of a directed graph* G with m nodes is a m x m matrix R = (r(i, j)), where r(i, j) is a 1 if and only if there is a path from node i and node j, otherwise the element is 0.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	1	1
5	0	0	0	0	0	1	1	1
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

Figure 7-2 Reachability matrix

It should be noted that while every digraph has a single reachability matrix, the inverse is not true there might be graphs which share one and the same reachability matrix, but are topologically different.

Warshall's Algorithm: A more efficient algorithm for calculating the reachability matrix was originally presented in [35], and is usually referred to in literature as Warshall's algorithm. From a programming point of view the algorithm is defined in the following way:

```
W = AdjacencyMatrix;
for (k = 0; k <= n - 1; k++){
   for (i = 0; i <= n - 1; i++)
   {
      for (j = 0; j <= n - 1; j++)
      {
         W(i, j) = W(i, j) | (W(i, k) & W(k, j));
      }
   }}
```

Warren's Algorithm An improvement over Warshall's algorithm has been proposed and proved in [34]. The algorithm is slightly more sophisticated, and looks as follows:

```
for (i = 2; i <= n; i++)
{
   for (j = 1; j <= i - 1; j++)
   {
      if (M(i, j) == true)
      {
         for (k = 0; k <= n - 1; k++)
         {
            M(i, k) = M(i, k) | M(j, k);
         }
      }
   }
}

for (i = 1; i <= n - 1; i++)
{
   for (j = i + 1; j <= n; j++)
   {
      if (M(i, j) == true)
      {
         for (k = 0; k <= n - 1; k++)
         {
            M(i, k) = M(i, k) | M(j, k);
         }
      }
   }
}
```

Partitioning Algorithm

In [22] it is stated that a popular DSM partitioning method has been presented in (Warfield, 1973). The algorithm presented is essentially a variation of the topology sorting of a digraph, with one important difference the author uses the reachability matrix to solve the dependency sequence, and cycles in the digraph are transparently processed without special treatment. For acyclic digraphs, the adjacency matrix is sufficient to partition the DSM correctly.

```
    private void PartitionDsm(BinaryMatrix ReachabilityMatrix)
    {
        int MatrixSize = ReachabilityMatrix.Count;
        SequenceCollection<int> ReachabilitySet = new
        SequenceCollection<int>(MatrixSize);
        SequenceCollection<int> AntecedentSet = new
        SequenceCollection<int>(MatrixSize);
        List<bool> ConsideredList = new List<bool>(MatrixSize);

        for (int i = 0; i <= MatrixSize - 1; i++)
        {
            ReachabilitySet.AddLevel(new List<int>());
            AntecedentSet.AddLevel(new List<int>());
            ConsideredList.Add(false);
        }

        for (int i = 0; i <= MatrixSize - 1; i++)
        {
            for (int j = 0; j <= MatrixSize - 1; j++)
            {
                if (ReachabilityMatrix(i, j) != 0)
                {
                    ReachabilitySet(i).Add(j);
                    AntecedentSet(j).Add(i);
                }
            }
        }

        int Unlabelled = MatrixSize;
        int CurrentLevel = 0;
        SequenceCollection<int> VarSequence = new SequenceCollection<int>();

        while (Unlabelled > 0)
        {
            VarSequence.AddNewLevel();

            for (int i = 0; i <= MatrixSize - 1; i++)
            {
                if (ConsideredList[i] == false &&
                NoDependencies(ReachabilitySet(i),
                AntecedentSet(i)) == true)
                {
                    VarSequence.AddToLevel(CurrentLevel, i);
                }
            }

            RemoveDependencies(ReachabilitySet,
            VarSequence.Level(CurrentLevel));
            RemoveDependencies(AntecedentSet, VarSequence.Level(CurrentLevel));

            foreach (int CurrentObject in VarSequence.Level(CurrentLevel))
            {
                ConsideredList[CurrentObject] = true;
            }

            Unlabelled = Unlabelled –
            VarSequence.CurrentLevelCount(CurrentLevel);
            CurrentLevel = CurrentLevel + 1;
        }
    }
}
```

1. Create a new partition level.
2. Calculate the reachability and antecedent sets R(s) and A(s).
3. For each element in the DSM, calculate the set product R(s)A(s).
4. If R(s)A(s) = R(s), add the elements to the current level.
5. Remove the elements from the list, and all references to it from the reachability and antecedent sets of all other elements.
6. Repeat from step 1, if the item list is not empty.

7.2 DSM Interface Component

The designed user interface manages interaction with the user for the purpose of partitioning the original matrix. The application acquires data from the user as a .cvs file and interprets events that are caused by user actions and finally displays the resulting partitioned DSM.

For the specific process *the ILNumerics.Net.dll* library is obtained from an open source project. The project source code is available from the link http://debris-kbe.sourceforge.net/.

Below sample screen shots of the application are given. The blue squares indicate a dependency that is satisfied by the current arrangement, while the red ones show when an item depends on another that appears after it in the current view. The alternating green and white background for the items in the DSM mark the separate levels into which the DSM has been partitioned [22].

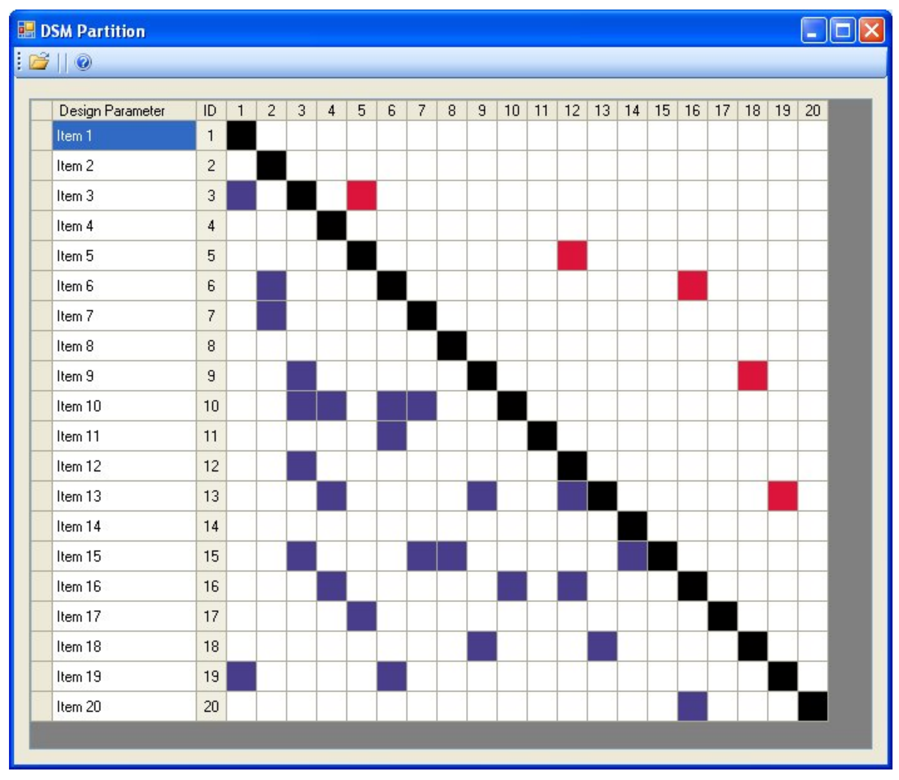


Figure 7-3 Original DSM

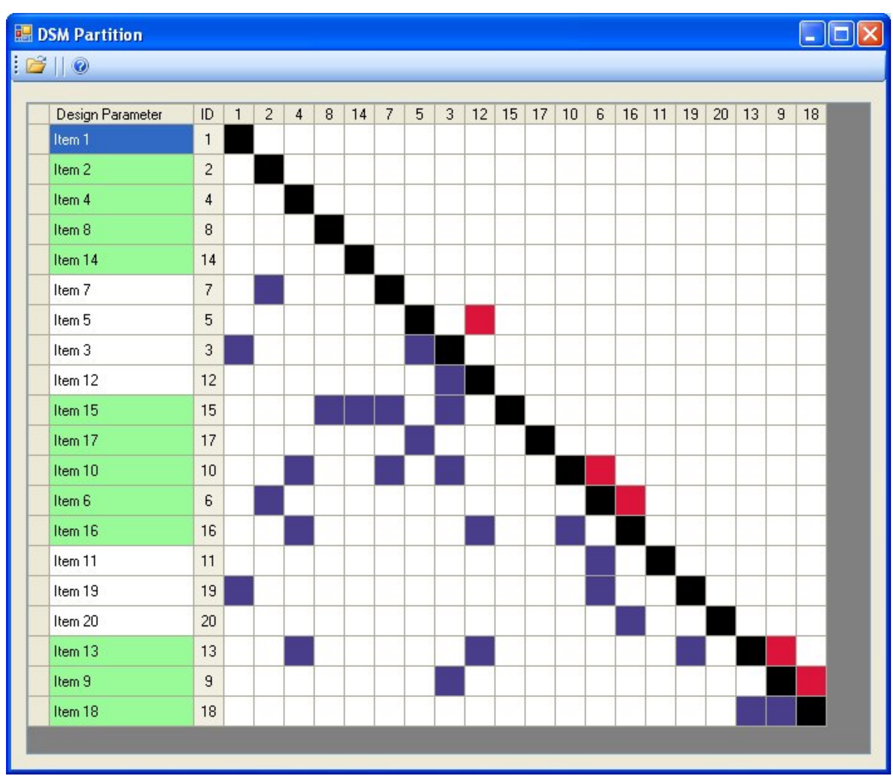


Figure 7-4 Optimized and partitioned DSM

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this study first, an introduction to dependency concept is worked out in great detail. At this stage the perspective of OO concepts with UML considerations are provided. In the second part of the study definitions of the terminology used for the DSM basics, DSM based algorithms, representation details and related works are provided. In the previous chapter a developed prototype is introduced with the main algorithms used for the partitioning.

The application of the DSM to software, with modules playing the role of tasks, is straightforward and yet appears to have several advantages over more widely used dependency representations. The matrix representation itself scales better than box- and line diagrams. The partitioning algorithms provide an automatic mechanism for architectural discovery in a large code base. Partitioning eliminates cycles by forming subsystems. The groupings and orderings recommended by these algorithms can be applied straightforwardly to reorganize the code base so that its inherent structure matches the desired structure. [24]

The main purpose of software engineering is to provide higher quality for the products. And it can be concluded that the identification of dependencies is important when scalability, performance, manageability, reuse, business context and granularity are considered.

The DSM method supports a major need in engineering design management: documenting information that is exchanged. The method provides visually powerful means for capturing, communicating, and organizing engineering design activities and architectural issues such as project team formation and product architecture. [37]

8.2 Future Work

An area of future development may be the implementation of interactive tools that is more flexible, with rule definitions and navigation properties. There are commercial and non commercial reflectors developed for analyzing .NET assemblies. A new DSM based component should be integrated with the current reflectors. For the .Net environment, it should analyze dependencies with the property of showing the source code which causes the dependency and representing dependency structure matrix automatically.

REFERENCES

[1] **Alda, S., Won, M., and Cremers, A. B.** (2003), *Managing Dependencies in Component-Based Distributed Applications,* In Revised Papers From the international Workshop on Scientific Engineering for Distributed Java Applications (November 28 - 29, 2002). N. Guelfi, E. Astesiano, and G. Reggio, Eds. Lecture Notes In Computer Science, vol. 2604. Springer-Verlag, London, 143-154.

[2] **Booch G, et. al.** (1998), *Unified Modeling Language User Guide*, Addison Wesley Massachusetts

[3] **Carimo, R.G.** , *Evaluation of UML Profile for Quality of Service from the User Perspective,* M.S School of Engineering Blekinge Institute of Technology, Ronneby Sweden

[4] **Chaves, L.F.C.P** (2008), *Deployment of Mobile Systems Using Clustering Techniques*, Federal University of Pernambuco Department of Computer Science

[5] **Cox, L., Delugach, H. S., and Skipper, D.** (2001), *Dependency Analysis Using Conceptual Graphs,* Proceedings of the 9th International Conference on Conceptual Structures, Palo Alto, CA

[6] **Cox, L., Delugach, H. S., and Skipper, D**., *Representing Software Component Dependencies Using Conceptual Graphs*

[7] **Donald V. Steward** (1981), *The Design Structure System: A Method for Managing the Design of Complex Systems,* IEEE Transactions on Engineering Management, 28(3):71-74

[8] **Eppinger**, et. al. (1994), *A Model-Based Method for Organizing Tasks in Product Development,* Research in Engineering Design 6 (1994): 1-13.

[9] **Garland J., Anthony R.** (2003), *Large-Scale Software architecture A Practical Guide using UML*, West SussexPO19

[10] **Heron, T.** (2002), *Programming with Dependency*, M.S, Department of Computer Science University of Warwick, Warwickshire, U.K

[11] http://en.wikipedia.org/wiki/Adjacency_matrix

[12] http://www.merriam-webster.com/dictionary/

[13] **Ian Graham , Alan Wills,** *UML Tutorial* Trireme International Ltd

[14] **J. Schmuller** (2004), *Teach Yourself UML in 24 Hours*, Third Edition, Sams Publishing

[15] **Karl-Linus Blomberg** et. al. (2005), *Mapping of Relations and Dependencies Using DSM/DMM-Analysis Casting Mold Manufacturing*, Internationel L A Handelshögskolan, Sweeden

[16] **Keller, A. Blumenthal, U. and Kar, G.** (2000), *Classification and Computation of Dependencies for Distributed Management,* Proceedings of the IEEE Symposium on Computers and Communication (ISCC 2000), IEEE Computer Society

[17] **Knublauch, H.** (2002), *An Agile Development Methodology for Knowledge-Based Systems Including a Java Framework for Knowledge Modeling and Appropriate Tool Support*, PhD. Thesis, Universty of Ulm

[18] **Krzysztof Czarnecki.** (Oct. 1998), *Generative Programming : Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, Ph. D. thesis, Technische Universitat Ilmenau, Germany.

[19] **L. A. Tuura**. (2003), *Ignominy: Tool for Analyzing Software Dependencies and for Reducing Complexity in Large Software System.* In Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research, volume 502, pages 684–686

[20] **L. Bixin.** (2003), *Managing Dependencies in Component-Based Systems Based on Matrix Model, In Proceedin*gs of NETObject-Days'03

[21] **MARTIN R.** (1994), *OO Design Quality Metrics - An Analysis of Dependencies,* Proc. of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA’94

[22] **Martin Tapankov** (2009), *Managing Dependencies In Knowledge-Based Systems: A Graph-Based Approach*, Thesis Work Product Development And Materials Engineering, Jönköping University, Sweeden

[23] **N. Wilde** (August 1990), *Understanding Program Dependencies, Software Engineering,* Institute Carnegie Mellon University, Report no: CM-26

[24] **Neeraj Sangal** et. al. (2005), *Using Dependency Models to Manage Complex Software Architecture,* In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, pages 167-176. ACM Press, New York, NY, USA

[25] **Rumbaugh J. et. al.** (1999), *The Unified Modeling Language Reference Manual,* ADDISON-WESLEY Massachusetts

[26] **Stevens, M.** (2007), *Design Structure Matrices for Software Development,* M.S Thesis, Faculty of Science Department of Computer Science, The Vrije Universiteit Brussel

[27] The Design Structure Matrix Web Site (DSM Community Site). http://www.dsmweb.org.

[28] The International DSM Conference. http://www.dsm-conference.org

[29] **Thomas A. Black** et. al. (1990), *A Method for Systems Design Using Precedence Relationships: An Application to Automotive Brake Systems,* Working Paper WP #3208-90-MS, Leaders for Manufacturing Program, MIT Sloan School of Management, Cambridge, MA, USA

[30] **Thomas W. Malone and Kevin Crowston.** (1994.), *The Interdisciplinary Study of Coordination,* ACM Comput. Surv., 26(1):87–119

[31] **Trigos, E.D.** (2009) ,*Service Dependency Analysis Based on Process Models and Service Level Agreement*, M.S Thesis, Dresden University of Technology, Dresten

[32] **Tyson R. Browning, Mike Danilovic** (2007), *Managing Complex Product Development Projects with Design Structure Matrices and Domain Mapping Matrices,* International Journal of Project Management 25 : 300–314

[33] **Tyson R. Browning**. (2001), *Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions.*, IEEE Transactions on Engineering Management, 48(3):292 – 306

[34] **Warren Jr., Henry S.** (1975), *A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations*, Communications of the ACM 18(4), 218-220.

[35] **Warshall, Stephen** (1962), *A Theorem on Boolean Matrices*, Journal of the ACM 9(1), 11-12.

[36] **Xinyi Dong** et.al, *System-level Usage Dependency Analysis of Object-Oriented Systems,* Software Architecture Group (SWAG), School of Computer Science University of Waterloo

[37] **Yassine, A.** (2004), *An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method,* Quaderni di Management, no. 9.

[38] **Zhang, Z.** (2009), *An Ontology-Based Reengineering Methodology for Service Orientation*, PhD. Thesis, Software Technology Research Laboratory, De Montfort University

APPENDIX

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Yücetürk, Oğuzhan
Nationality: Turkish (TC)
Date and Place of Birth: 19 April 1975, Çorum
Marital Status: Married
Phone: +90 312 289 12 09
email: oguzhan.yuceturk@gmail.com

EDUCATION

Degree	Institution	Year of Graduation
MS	Çankaya Univ.Computer Engineering, Ankara	2010
BS	METU Electrical and Electronics Engineering, Ankara	1998
High School	Kayseri Science High School, Kayseri	1993

WORK EXPERIENCE

Year	Place	Enrollment
2004 - Present	FINTEK	Senior Software Engineer
2002 - 2004	TUBITAK MAM	Researcher
2000 - 2002	ERICSSON	Network Configuration Engineer
1998 - 1999	PAMUKBANK	Software Engineer

FOREIGN LANGUAGES

Advanced English

COMPUTER SKILLS

System Knowledge	: Windows XP
Programming Languages	: C,C++, C#, Java, PHP, HTML
Application Software	: Visual Studio.Net, Borland Builder, Rational Rose RT, Visio, Clear Case, Visual Source Safe, MS Office Applications, Business Objects Reporting Tool, Reflector
Database Applications	: Oracle 10g, PL SQL Developer 7.0, MS Access

HOBBIES

Football, movies, reading.