

**PERFORMING STATIC STRUCTURE ANALYSIS
USING SOFTWARE DEPENDENCIES**

by

Ashgan Fararooy

B.Sc. Mathematics, Sharif University of Technology, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Ashgan Fararooy 2010
SIMON FRASER UNIVERSITY
Spring 2010

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Ashgan Fararooy
Degree: Master of Science
Title of Thesis: Performing Static Structure Analysis Using Software Dependencies

Examining Committee: Dr. Janice Regan
Chair

Dr. Dirk Beyer,
Assistant Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Torsten Möller,
Associate Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Uwe Glässer,
Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: April 14, 2010



SIMON FRASER UNIVERSITY
LIBRARY

Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <www.lib.sfu.ca> at: <<http://ir.lib.sfu.ca/handle/1892/112>>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Abstract

Software quality assessment and program comprehension have been challenging areas of research in software engineering. Software dependencies bear valuable information that can be utilized to gain insight into computer programs and compare different program versions. We present a simple and effective indicator for structural problems and complex dependencies on code-level, together with an automatic monitoring tool. We model low-level dependencies between program operations using a use-def graph, which is generated from reaching definitions of variables. Intuitively, a program operation that has more dependencies is harder to understand because it requires consideration of more elements and possibilities. Using various examples we show that the proposed analysis can be a good indicator of readability and understandability of programs. We also developed another tool that inspects dependencies on the architecture level. The tool visualizes introduced and removed dependencies across different program versions.

Keywords: Static analysis; Readability; Refactoring; Product metrics; Software visualization

To my parents

“No self-respecting architect leaves the scaffolding in place after completing the building.”

Carl Friedrich Gauss, 1777-1855

Acknowledgments

I am glad to have had the opportunity to work with my senior supervisor, Dr. Dirk Beyer, and I am grateful for everything that I learned from him. I would like to thank my supervisor, Dr. Torsten Möller, for the interesting and inspiring discussions during which I received many valuable comments and suggestions. I am grateful to Dr. Uwe Glässer for his constructive comments and feedback that helped to improve the final document.

I would also like to express my gratitudes to Dr. Janice Regan for chairing my defense, Erkan Keremoglu for the helpful discussions and sharing his ideas, Masoud Nosrati for his endless help and support, Sepehr Attarchi for helping me prepare and improve the defense presentation, and all my supportive friends who made this possible.

I truly appreciate the support of my partner, Renata Ruffell, and her continued patience throughout my studies. And most of all, my deepest thanks goes to my family, who had a major role in helping me achieve this.

Contents

Approval	ii
Abstract	iii
Dedication	iv
Quotation	v
Acknowledgments	vi
Contents	vii
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Control-Flow Graphs	3
1.2 Software Dependencies	4
1.3 Thesis Outline	4
2 Related Work	6
2.1 Overview	6
2.2 Size Indicators	7
2.3 Complexity Indicators	7
2.4 Halstead's Measures	8
2.5 Quality Indicators	9

2.6	Summary	10
I	The Measure Dep-Degree	11
3	Definitions	12
3.1	Preliminaries	12
3.1.1	Control-Flow Graph (CFG)	12
3.1.2	Reaching Definitions	13
3.1.3	Use-Def Graph	13
3.2	An Indicator for Problematic Code Structure	14
3.2.1	Dependency-Degree	14
3.2.2	Problematic Code Structure	15
4	Exploring Dep-Degree	17
4.1	Assignments and Arithmetics	17
4.2	Strength Reduction and Nested Loops	18
4.3	Early Return	20
5	Applications of Dep-Degree	24
5.1	Assessment before and after Refactoring	24
5.1.1	Extract Method	25
5.1.2	Inline Method	26
5.1.3	Introduce Parameter Object	28
5.1.4	Parameterize Method	30
5.1.5	Pull Up Method	30
5.1.6	Replace Conditional with Polymorphism	32
5.1.7	Revisions from Code Repositories	33
5.1.8	Position objects instead of 3-dim arrays	37
5.1.9	Summary of Results	38
5.1.10	Possible Limitations	38
5.2	Indication of Problematic Code	39
5.2.1	Identifying problematic operations	39
5.2.2	Identifying problematic functions	40

6	Tool Implementation	43
6.1	Algorithm	43
6.2	Features and Architecture	44
II	Dependency Analysis on Architecture Level	46
7	Overview	47
8	Features	50
9	Applications of CheckDep	53
9.1	Development	53
9.2	Refactoring	53
9.3	Structure Assessment	54
9.4	Design Change Identification	55
9.5	Subversion Dependency Report	55
10	Conclusion and Future Work	57
10.1	Conclusion	57
10.2	Future Work	58
	Bibliography	59

List of Tables

4.1	The values of the indicators LOC (lines of code), CC (cyclomatic complexity), DD (dep-degree), and Halstead’s HD (Difficulty) and HE (Effort) for the three examples	20
5.1	The ‘Extract Method’ example before and after refactoring	25
5.2	The ‘Inline Method’ example before and after refactoring	28
5.3	The ‘Parameter Object’ example before and after refactoring	29
5.4	The ‘Parameterize Method’ example before and after refactoring	31
5.5	The ‘Pull Up Method’ example before and after refactoring	33
5.6	The ‘Replace Conditional with Polymorphism’ example before and after refactoring	33
5.7	Refactoring ‘parseCmdLine’ from r28 to r34	36
5.8	Enum change from r20 to r21	36
5.9	Better color handling from r37 to r38	36
5.10	Improvements from r43 to r44 and from r117 to r118	37
5.11	Improved Position from r112 to r117	41
8.1	Tool performance on open-source projects (all times are given in seconds) . .	52

List of Figures

1.1	The control-flow graph for Example 1	3
3.1	Control-Flow Graph for Example 2	14
3.2	Searching for Reaching Definitions	14
3.3	Use-Def Graph for Example 2	15
4.1	Two ‘swap’ implementations	18
4.2	Use-def graphs for Figure 4.1	18
4.3	Two ‘bico’ implementations	19
4.4	Use-def graph for Figure 4.3 (left)	21
4.5	Use-def graph for Figure 4.3 (right)	21
4.6	Use-def graph for Figure 4.8 (left)	22
4.7	Use-def graph for Figure 4.8 (right)	22
4.8	Two ‘equals’ implementations	23
5.1	An Example of ‘Extract Method’ Refactoring	27
5.2	An Example of ‘Inline Method’ Refactoring	29
5.3	An Example of ‘Introduce Parameter Object’ Refactoring	30
5.4	An Example of ‘Parameterize Method’ Refactoring	32
5.5	An Example of ‘Pull Up Method’ Refactoring	34
5.6	An Example of ‘Replace Conditional with Polymorphism’ Refactoring	35
5.7	Markers locating operations with highest dep-degree values	40
5.8	Coloring program operations according to their dep-degree values	42
8.1	CheckDep Architecture	52

9.1	A ‘pull-up method’ refactoring removes 6 dependencies (green) and adds 3 (red), which improves the software structure	54
9.2	Localizing dependency changes; discs represent software artifacts (e.g. methods); the rectangle indicates a zoom area, containing most of the changed artifacts which are colored in red or green.	56

Chapter 1

Introduction

In software engineering, static analysis refers to the practice of inspecting and analyzing computer programs without executing them. This type of analysis is usually performed on the source code of a program, and in some cases on the object files (compiled units). Static analysis can be used to gain more insight about the program code and structure, to detect software defects and anomalies, to provide information for proving properties of a computer program (verification) and to facilitate the decision making procedure in a software process. In most cases, static analysis denotes automated processes or procedures which can be performed by software-analysis tools, possibly yielding more accurate results than those of human experts, in less time.

Program analysis can provide directions for improving different aspects of software quality that affect customer satisfaction. It also helps decrease maintenance costs which constitute a significant proportion of the full life-cycle cost of a software system [1, 21]. The importance of such analyses is even more visible if we consider safety-critical systems such as process-control systems in pharmaceutical plants, or control and monitoring systems in aircrafts which are operated by complicated programs consisting of millions of lines of code. Malfunctioning of such software might result in hardware failure which can be a big threat to human lives and the environment [34].

Many desirable software qualities have been defined and studied. Coming up with quantifiable measures to assess these quality factors for different software systems is one of the challenges of the software community due to the complexity of software systems. However, a lack of such measures means subjective and often unreliable opinion would have to be used instead, which is against the fundamental goals of engineering.

Software systems, due to the frequency and amount of changes to their structure, are very different from artifacts in other engineering disciplines. The frequent changes – often essentially affecting the stability of the system – require a continuous effort to prevent the structure from degeneration. There are several theories why this must happen (e.g., [23]) and several proposals and guidelines on how to prevent or fix this (e.g., [9, 10, 29]).

We present a simple but effective idea that contributes to solving the following subproblem: Given two versions of a software program that have the same behavior and differ only in code structure, which version is to prefer in terms of low-level dependency structure. For example, if the second version is the result of changing the first version, we would like to know whether the change actually improves the code structure, i.e., was a positive ‘refactoring’. This is achieved by introducing a new software indicator, which reflects the quality degree of the software with respect to a specific quality attribute.

We have the following requirements that the new indicator has to fulfill. Simplicity (easy to interpret), scalability (applicability to different versions), and independence (complementing other indicators) are inspired by Hansen’s list of requirements for software-complexity measures [14].

1. **Simplicity.** We are striving to simplify code, and believe this requires a simple method. Therefore, the indicator must be easy to understand and the indicator values easy to interpret.
2. **Flexibility.** We are looking for an indicator that works for many imperative programming languages.
3. **Scalability.** The new indicator should be applicable to partial programs (e.g., to be applicable to diffs), should not require structured code, or the presence of the complete control-flow structure.
4. **Independence.** The indicator should not be based on existing indicators, and should be as orthogonal and complementing as possible.
5. **Automatic.** The calculation of the indicator values should be based only on facts present in the program code, and not require experience or intelligence.

To further pursue our program comprehension goals, we also describe a new software tool that we developed to track and visualize software artifacts and their changes during

software evolution. Due to the nature of the material presented here, they are categorized as static analysis activities and techniques.

Some software-engineering concepts used in this report are briefly discussed in the following sections of this chapter.

1.1 Control-Flow Graphs

A control-flow graph is a representation of a computer program using directed graphs. Some nodes in the graph represent one or more procedural statements, i.e. basic blocks containing a single or multiple sequential instructions [25]. Some nodes do not represent any procedural statements and are meant as ‘auxiliary’ nodes to direct the flow to the next step. Edges (links) of the control flow graph represent flow of control, and depict a jump from one block (node) in the program to the next one. A predicate node is one that contains a condition from which two or more edges emanate. Each subroutine (i.e., function, method or procedure) is assumed to have a single ‘entry’ node and an ‘exit’ node.

Control-flow graphs are used by many static analysis tools. Example 1 shows a simple Java method which calculates the greatest common divisor of two input integers ($n > m > 0$), and Figure 1.1 depicts the respective control flow graph:

```
int gcd(int n, int m)
{
  int q = n, r = m;
  while (r != 0)
  {
    int temp = q % r;
    q = r;
    r = temp;
  }
  return q;
}
```

Example 1

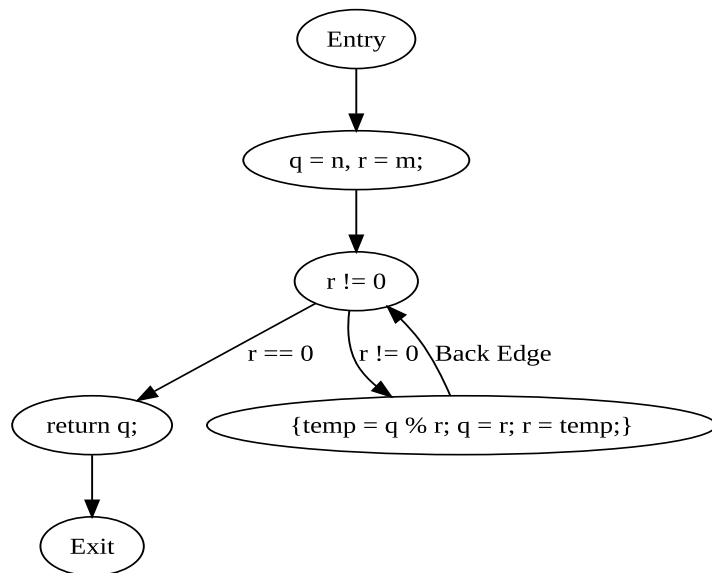


Figure 1.1: The control-flow graph for Example 1

1.2 Software Dependencies

The term software dependencies can have different meanings in different contexts. It could refer to, for example, the relations (dependencies) between a software package/component and others, in the same or different versions of the software.

There might be different strategies for defining dependencies between software artifacts, depending on the desired type of analysis and the required information.

In object-oriented programming, dependencies can be defined in terms of certain relationships between the elements of this programming paradigm, such as inheritance relations between classes. A method-call dependency can be defined as the dependency between two methods (functions) where one invokes (calls) the other in its body. Similarly, a class-call dependency results when a method of one class calls a method which belongs to another class. Then we say that the former is dependent on the latter. We can have dependency between packages as well, when a class in one package is dependent on a class from another one. Another type of dependency that can be defined between the classes is when a class is dependent on another one for having a data field of the type specified by the latter.

On a lower level of abstraction, we can define dependencies between ‘fine-grained’ software items such as lines of codes or even atomic program statements. One way to do this is based on ‘use-def’ relations between variables within program statements. In other words, a statement that uses a variable can depend on a statement that defines (assigns a value to) it. This kind of dependency is more general and is applicable to programs written in other programming paradigms or even unstructured code.

1.3 Thesis Outline

We proceed as follows: In Chapter 2, we discuss related work from the area of software measurement. Chapter 3 presents the formal definition of a new software indicator called ‘dependency-degree’ (or for short ‘dep-degree’), after introducing preliminaries. Chapter 4 explores the proposed definition through some insightful examples. Chapter 5 discusses applications of the indicator. First, we apply dep-degree to several sets of refactorings and compare the indicator values with those of LOC (lines of code) and cyclomatic complexity. Second, we suggest to apply the indicator in order to identify and locate code with complex dependency structure. Chapter 6 presents a software tool that is implemented to calculate

the indicator for java code, the algorithm used in the tool and its computational complexity. Chapter 7 presents another plug-in tool that we developed to visualize and track introduced and removed dependencies (on a more abstract level than code-level) across different versions of the software. Chapter 8 concludes with an overview of the presented materials and discusses possible future contributions.

Chapter 2

Related Work

Measurement plays a major role in any engineering process by attempting to provide a better understanding of the attributes of the systems and models under study. Broadly speaking, the measurement can be defined as “the assignment of numerals to objects or events according to rules” [36]. But there are a variety of ‘rules’ under which these assignments can be applied, yielding different kinds of scales and measurements. In order for an indicator to be considered a measure, certain details such as the type of scale should be clarified.

The concept of measurement has not yet been settled in software engineering as it has in some other disciplines. In most cases, there is no way to measure software quality attributes directly [34]. For example external attributes such as ‘understandability’ and ‘maintainability’ are related to how developers view the software and depend on many factors, making it a difficult task to measure them. That is why an internal attribute of software (such as its length) is measured instead, and it is assumed that a relationship exist between what can be measured and what we really want to know.

2.1 Overview

There is a rich set of software measures defined in the literature to evaluate different aspects of software systems. Some of these measures are designed only for specific programming paradigms such as object-oriented programming [24, 30, 38]. There are numerous classic software measures that have been studied extensively throughout the years, and some of them are still widely in use today. However, recent efforts have been made to create software

measures supported by a richer theoretical background [20].

In a survey by Mills [27], classic software measures are generally classified as either product measures or process measures. Process measures are associated with particular models of software development process and are used to measure different aspects of the process such as the overall development time or the average level of experience of the programming staff. Process measures are not the concern of this study.

Product measures, on the other hand, are designed to assess various attributes of a software product at any stage of its development. They may be categorized into four types, namely size measures, complexity measures, Halstead's product measures and quality measures [27]. Some of these categories may overlap; for example some complexity measures might be used to evaluate a certain quality characteristic of software.

2.2 Size Indicators

Perhaps the simplest proposed measure is the lines of code (LOC), an indicator for the size of a program. It measures the length of a program by a pure syntactical count of lines, without analyzing the contents in detail. LOC can be calculated in a number of different ways, for examples when considering only non-comment and non-blank lines in a source code, it is called the 'logical lines of code'. Despite its popularity, LOC is invalid under various conditions, for example when it is "used to compare productivity or quality data across different programming languages" [18].

2.3 Complexity Indicators

Complexity measures are designed with the intent to measure 'program complexity'. A prominent complexity measure is the 'cyclomatic complexity' [25]. It is an indicator for the complexity of the control-flow structure of a program because it measures the maximum number of linearly independent paths in the control flow graph of a computer program. It is defined as $v(G) = e - n + 2p$, where e , n and p are the number of edges, vertices and the connected components of the control flow graph respectively.

The cyclomatic complexity measure has been found to generate useful information [18]. Nevertheless, it has certain limitations. For example it is simply an indicator of a program's structural complexity and not its data flow complexity. Also, it assigns the same complexity

value to an if-else statement and a while loop, even though in practice a loop is more complex than an alternative. Another limitation of the cyclomatic complexity is that it does not distinguish between nested and sequential forms of control structures.

There have been efforts to improve existing measures such as the cyclomatic complexity. Examples include Myer's suggestion to remove an undesirable anomaly from the original definition, or Stettar's extension to include data declarations and references in the program flow graph [28, 35]. However the extensions also have some limitations. For example, in practice the Myers' variant is not significantly different from the original value as computed by McCabe [11]. Another variant of the cyclomatic complexity is the 'essential complexity'. The essential complexity of any program that is written entirely with high-level 'structured programming' language constructs will be one. This measure is useful for evaluating the complexity of an unstructured code, or detecting an implementation of poorly structured logic.

A measure of the logical complexity of programs has also been proposed based on the variable dependency of sequence of computations, inductive effort in writing loops and complexity of data structures [17]. A graph is used to describe the dependence of a computation at a node upon the computation of other nodes.

Some proposed to measure program complexity based on data-flow information. For example Kafura and Henry proposed a measure of local information flows entering (fan-in) and exiting (fan-out) each procedure (function) [15]. Tai proposed a measurement approach based on data flow information in a control flow graph [37]. His measure is defined for a structured control-flow graph, provided that certain conditions are met.

2.4 Halstead's Measures

Halstead's measures [12] are also often studied as possible measures of software complexity [27]. Halstead considers a computer program as a collection of tokens that can be classified as either operators or operands. The following base measures are defined by Halstead [31]:

- n_1 : The number of distinct operators
- n_2 : The number of distinct operands
- N_1 : The total number of occurrences of operators
- N_2 : The total number of occurrences of operands
- n_1^* : The number of potential operators
- n_2^* : The number of potential operands

All the other measures defined by Halstead are based on the above measures. For example program length and program vocabulary are defined in the following manner:

$$\text{Program length } (N): \quad N = N_1 + N_2$$

$$\text{Program vocabulary}(n): \quad n = n_1 + n_2$$

According to Al Qutaish and Arban, Halstead defines program volume as (a) a suitable measure for the size of any implementation of any algorithm; (b) a count of the number of mental comparisons required to generate a program [31].

Since some of Halstead's measures are not defined in a computable way, practitioners (e.g., NASA¹) have used the following approximations:

$$\text{Program Volume } (V): \quad V = N \times \log_2 n$$

$$\text{Difficulty } (HD): \quad HD = \frac{n_1}{2} \times \frac{N_2}{n_2}$$

$$\text{Effort } (HE): \quad HE = V \times HD$$

Halstead's 'difficulty' indicator (HD) can suggest the difficulty level of a program, and the 'effort' indicator (HE) indicates the estimated effort required to develop a program.

2.5 Quality Indicators

Quality measures target the assessment of software quality factors such as program correctness, reliability and maintainability. For example, the number of defects in a software product could be an indicator for the correctness of software. Such measures are often extracted automatically from the product, e.g. the number of errors detected in program tests. As mentioned before, there have been investigations of the ability of some complexity measures such as Halstead's effort and cyclomatic complexity by McCabe to predict the

¹http://mdp.ivv.nasa.gov/halstead_metrics.html, as retrieved on April 27, 2010.

psychological complexity of software maintenance tasks [8].

2.6 Summary

The indicators measure certain properties of software, trying to indicate size, product properties, quality, and complexity. For our comparison, we chose LOC as the most prominent indicator for size, cyclomatic complexity as the most prominent indicator for control-flow complexity, and the Halstead's difficulty and effort indicators. For details about the various measures we refer the reader to the survey and discussion articles on software measures [7, 8, 15, 16, 18, 19, 27].

The application of software measures has significantly advanced the techniques to automatically and abstractly assess properties of large software systems. But many software engineers were too enthusiastic in applying measures, trying to use measures as indicators for properties that they were not designed for. For example, LOC was often considered a measure for size, but it is a measure for length, and just an indicator for size. Or, cyclomatic complexity was sometimes used as measure for program complexity, but it is a measure for cyclomatic complexity of control-flow graphs, and might only roughly indicate program complexity.

In order to compare our indicator dep-degree with other measures, we choose a few widely used (but not necessarily accepted) measures for software programs, i.e. LOC, cyclomatic complexity (CC) and Halstead's difficulty (HD) and effort (HE). We tried to apply Tai's measure [37] in our experiments as well, but it is not possible to calculate the measurement value due to several limiting requirements under which the value is defined.

Part I

The Measure Dep-Degree

Chapter 3

Definitions

In this chapter we provide the formal definitions of several concepts that are used in our study.

3.1 Preliminaries

Our proposed software indicator is based on some prerequisite concepts, the definitions of which are given within this section.

3.1.1 Control-Flow Graph (CFG)

We represent a computer program as a collection of control-flow graphs [2], one for each function (or procedure) of the program. A control-flow graph $G = (B, F)$ is a directed graph that consists of a set B of program operations (the nodes of the graph) and a set $F \subseteq B \times B$ of control-flow edges of the program. A program operation is executed when control moves from the entry to an exit of the operation node. A program operation is either an assignment operation, a conditional, a function call, or a function return. A conditional is a predicate that must be evaluated to true (false) for control to proceed along the first (second, resp.) exit edge. All other operations have one exit edge. Program operations can read and write values via variables from the set X of program variables.

In classic compiler literature (e.g., [2]), the nodes represent basic blocks, i.e., sequences of operations, as illustrated in the control-flow graph subsection in the introduction section. In our definition of CFG, a node represents only one program operation, as often done in

program analysis. The two variants of CFGs are equivalent in terms of program semantics, and can be transformed into each other.

3.1.2 Reaching Definitions

In this paper, we use a notion of operation dependency that is motivated by the data-flow analysis for reaching definitions [2]. The function *reaching definitions* $rd_G : B \times X \rightarrow 2^B$ for a CFG $G = (B, F)$ assigns to a program operation b_u and a variable x the set of all definitions of variable x that can reach the operation b_u . In other words, a program operation b_d is in the set of reaching definitions for program operation b_u and variable x , if b_d is an assignment operation or a function call that assigns a value to x and there exists a path in the CFG from b_d to b_u on which no other program operation assigns a value to x .

Example 2 shows a simple function, the control-flow graph of which is illustrated in Figure 3.1. Figure 3.2 shows how the reaching definitions of the last program operation (the return statement) with respect to the only variable used by the operation (i.e. ‘*min*’) can be identified. In this case, two reaching definitions (operations) are detected, that is $\{ \text{‘min = m’}, \text{‘min = n’} \}$. Note that the initializer statement ‘*int min = 0*’ is not a reaching definition for the return statement, because the value of the variable ‘*min*’ is updated in both branches of the ‘if’ statement.

3.1.3 Use-Def Graph

We now derive the use-def graph from the results of the reaching-definitions analysis. A *use-def graph* $S_G = (B, E)$ for a CFG $G = (B, F)$ is a directed graph that consists of the set B of program operations (of G) and the set E of use-def edges that are derived from the reaching-definitions function as follows: an edge (b_u, b_d) is member of the set E if there exists a variable x that is used in b_u and for which $b_d \in rd_G(b_u, x)$ holds.

The use-def graph is a *dependency graph* on operation level, more precisely, it models the data-flow dependencies between operations and the direction of an edge models the direction of the dependency (from use to definition). In compiler optimization and program analysis, this data-flow dependency is one of the most important dependencies that are considered (but mostly stored in a different form as so-called ud-chains [2]).

```

int
minimum(int m, int n)
{
    int min = 0;
    if (m < n)
        min = m;
    else
        min = n;
    return min;
}
    
```

Example 2

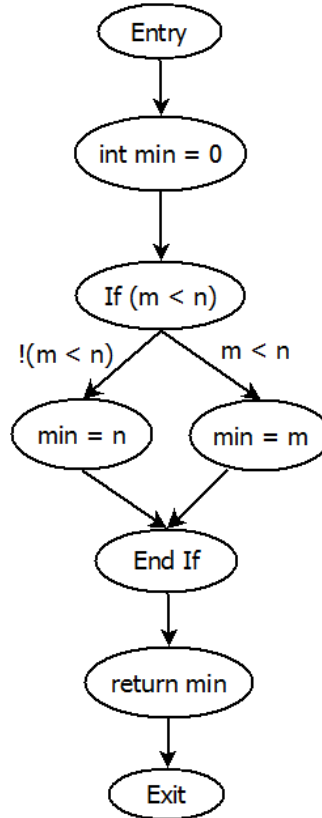


Figure 3.1: Control-Flow Graph for Example 2

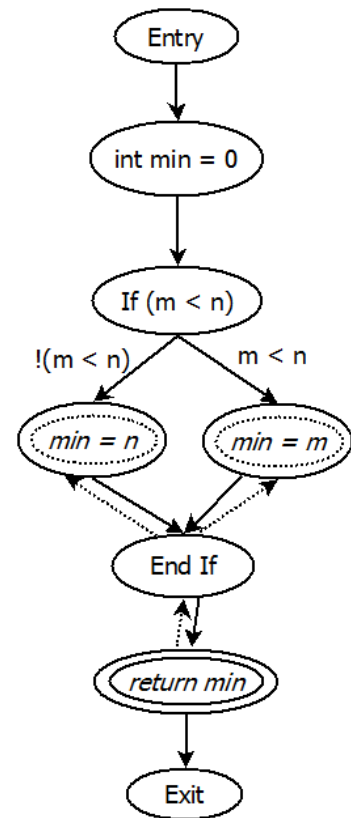


Figure 3.2: Searching for Reaching Definitions

Figure 3.3 shows the use-def graph for the Example 2 discussed in the previous subsection. A node labeled ‘init def’ refers to the parameter initialization.

3.2 An Indicator for Problematic Code Structure

This sections contains the formal definition of the new software indicator that we are proposing. The proposed indicator has also been discussed in a related publication [5].

3.2.1 Dependency-Degree

The *dep-degree for program operations* in a CFG $G = (B, F)$ is a total function $dd_G : B \rightarrow \mathbb{N}$ that assigns to each program operation b the number of other program operations that it depends on in $S_G = (B, E)$, i.e., $dd_G(b) = |\{b' \in B \mid (b, b') \in E\}|$ (the out-degree of b in

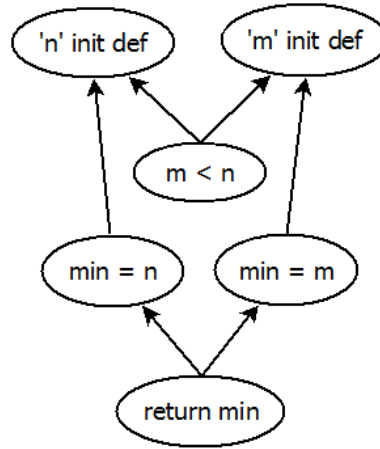


Figure 3.3: Use-Def Graph for Example 2

graph S_G). The *dep-degree for program functions* is a total function $dd : \mathbb{G} \rightarrow \mathbb{N}$ that assigns to each control-flow graph G the number of edges in its dependency graph $S_G = (G, E)$, i.e., $dd(G) = \sum_{b \in B} dd_G(b) = |E|$ (the sum of all out-degrees in graph S_G).

3.2.2 Problematic Code Structure

Inspired by Miller’s article on our capacity for processing information [26], we believe that the comprehension of program code is easy if we have to remember only a few possible states of the program (e.g., different variable values, branching choices), and that we make mistakes while programming, or misunderstand a program, if we have to remember too much information about the current program state.

The dep-degree for a single program operation tells us how many different pieces of information we need to consider in order to understand the effect of the program operation; more precisely, it tells us the number of all different predecessor operations that influence the effect of the considered program operation (it sums up, over all variables used in the operation, the number of different reaching definitions). Thus, if Miller’s insight is true for program understanding, then the dep-degree of an operation is a good indicator for the difficulty to understand the operation. It should be noted that we do not refer to dep-degree as a measure for code complexity. There is no established empirical relationship between software artifacts that is called code complexity. There are several attempts (e.g., McCabe, Halstead), but none is generally accepted as a measure for code complexity.

Remarks about dep-degree. We now provide some remarks on how dep-degree meets our requirements from the introduction:

1. **Simplicity.** The concept of reaching definitions is well-known, and the construction of the use-def graph as well as the calculation of the dep-degree value are straightforward. Thus, dep-degree can be implemented using efficient standard techniques from data-flow analysis.
2. **Flexibility.** Dep-degree is applicable to any imperative programming language. And yet, due to its simplicity, it has the potential to be customized to address certain language-specific issues.
3. **Scalability.** Dep-degree is applicable to well-structured (for, while, no goto, no break) as well as unstructured program code. It is applicable to complete functions as well as partial programs (for partial programs, the value of dep-degree is based on the reaching definitions that are present in the code fragment).
4. **Independence.** Dep-degree is a base indicator, i.e., it is solely based on facts present in the code, it does not use any arithmetics or combination with other indicators. Many other indicators are based on the control-flow structure, while dep-degree is exclusively based on the data-flow structure; the indicator is a good complement to other indicators.
5. **Automatic.** The calculation of the value for dep-degree consists of simply counting edges in the use-def graph, for certain operations or the full graph. The use-def graph is generated from reaching definitions, which can be computed in polynomial time using standard program-analysis techniques.

Chapter 4

Exploring Dep-Degree

This chapter intends to provide the reader with some insight into our proposed analysis using some small but rich examples.

4.1 Assignments and Arithmetics

Consider the two implementations of the function `swap` in Figure 4.1. The first implementation (left) has the advantage of using only two registers – which are allocated already anyway – but the disadvantage of being more difficult to understand because it uses not only assignments but also arithmetics.¹ The second implementation (right) has the advantage of being easy to understand – it uses only assignment operations – but the disadvantage that a simple code generator would allocate three registers for the execution of this code.

Figure 4.2 shows the use-def graphs for the two `swap` functions (a node labeled ‘init def’ refers to the parameter initialization of the call-by-value). The graph layout was calculated using `GRAPHVIZ` (`dot`). On the right, the value of variable `b` (third assignment) depends on the assignment of variable `temp` which in turn depends on the initial value of variable `a`. The value of variable `a` depends on the initial value of variable `b`. The graph on the left illustrates that this implementation not only involves arithmetics, but also has a more complicated dependency structure.

The dep-degree is six for the function on the left and three for the function on the right, which indicates that the function on the left has a more complex dependency structure.

¹Furthermore, one has to understand the arithmetic-overflow semantics of the programming language in order to establish correctness.

```

void
swap(int a, int b) {
    a += b;
    b = a - b;
    a -= b;
}

```

```

void
swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

```

Figure 4.1: Two ‘swap’ implementations

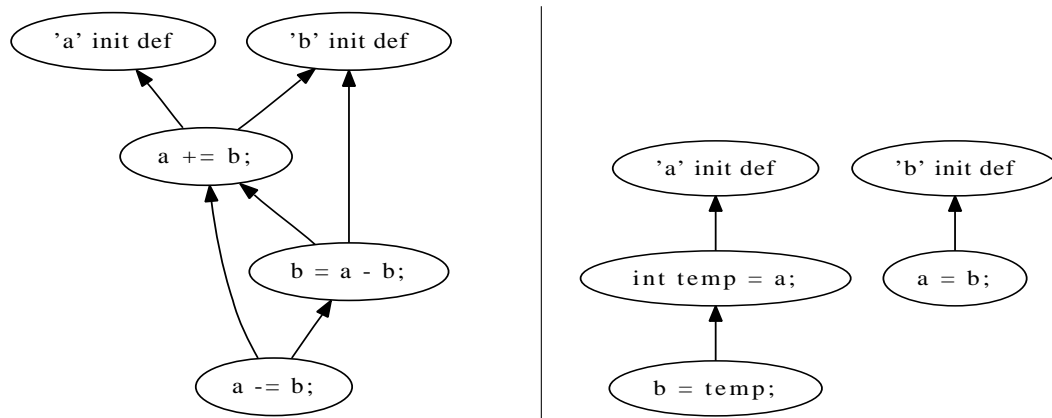


Figure 4.2: Use-def graphs for Figure 4.1

Table 4.1 shows that LOC and cyclomatic complexity do not distinguish the two functions. This is because LOC measures length and McCabe measures control-flow complexity, which is the same for both functions. The difficulty indicator (Halstead’s difficulty) suggests that the function on the left has a higher ‘difficulty’ level than the one on the right. The value of the effort indicator (Halstead’s effort) for the left function is also higher than the one for the right function.

4.2 Strength Reduction and Nested Loops

In Figure 4.3 we compare two implementations for computing binomial coefficients. Both functions take as input two non-negative integers n and k (required: $k \leq n$), and compute the binomial coefficient $\binom{n}{k}$ (n choose k).

The function on the left computes the result without using multiplication — it simulates

<pre> // Require: n >= k >= 0 int bico(int n, int k) { int[] arr = arrInit(n+1); for (int i = 0; i <= n; i++) { int temp = arr[0]; for (int j = 1; j < i; j++) { arr[j] = arr[j] + temp; temp = arr[j] - temp; } } return arr[k]; } </pre>	<pre> // Require: n >= k >= 0 int bico(int n, int k) { int facK = 1; for (int i = 1; i <= k; i++) { facK = facK * i; } int facNk = 1; for (int j = n; j > n-k; j--) { facNk = facNk * j; } return facNk / facK; } </pre>
--	--

Figure 4.3: Two ‘bico’ implementations

Pascal’s triangle to perform the computation. The array `arr` contains the i -th row of the triangle at the end of the i -th iteration of the outer ‘for’ loop. The disadvantage of this program is that it is rather difficult to understand because it uses nested loops instead of a sequence of two loops, and it uses an array, the content of which is important to understand. (An array access is more difficult than a variable access because it involves the array pointer and an index.)

The function on the right computes (almost directly) the result using the formula $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, but has the disadvantage of using multiplication (more expensive to compute, more expensive to verify because not linear). We say ‘almost directly’ because the second ‘for’ loop calculates $n(n-1)\dots(n-k+1) = \frac{n!}{(n-k)!}$.

The two functions `bico` are equal in the number of lines of code, the number of statements, and the number of local variables (`i, j, temp, arr` versus `i, j, facK, facNk`). Therefore, the length of the functions LOC yields the same value for both functions. Furthermore, the functions use the same number of control structures (two ‘for’ loops), and therefore the cyclomatic complexity yields the same value for both functions. But the low-level dependency structures of the two functions are very different. The dependency graphs are shown in Figures 4.4 and 4.5. The graph in Figure 4.4 has higher density such that the graph-drawing algorithm ‘dot’ from GRAPHVIZ was not able to find a layout without edge crossings. Table 4.1 lists the values for LOC, cyclomatic complexity, and dep-degree. LOC and cyclomatic complexity yield the same values, whereas dep-degree yields the values 28 and 24 for the left and right implementations respectively, indicating that the first implementation has a more complex dependency structure. Unlike dep-degree, both Halstead’s indicators

Method	LOC	CC	DD	HD	HE
swap (left)	3	1	6	8.8	343.9
swap (right)	3	1	3	3.0	100.8
bico (left)	9	3	28	18.7	4919.4
bico (right)	9	3	24	21.4	5076.7
equals (left)	10	3	11	13.3	2694.0
equals (right)	11	4	8	8.5	1453.2

Table 4.1: The values of the indicators LOC (lines of code), CC (cyclomatic complexity), DD (dep-degree), and Halstead’s HD (Difficulty) and HE (Effort) for the three examples

(difficulty and effort) have smaller values for the left implementation.

4.3 Early Return

In the next example we consider two alternative implementations of the `equals` function for a class `Pair` (of two integer values). Figure 4.8 shows the example functions. The two functions follow the same logic, but the first implementation uses a local variable `result` to store the decision to return, whereas the second implementation returns as early as possible.

The second implementation seems to be easier to understand, because all special cases are checked and immediately dealt with; after this, the reader can forget them, i.e., there are not many dependencies. The first implementation requires the reader to track the outcome of the various comparisons, and the last value of variable `result`, all the way to the end of the function.

The cyclomatic complexity of the second implementation is higher, because it uses one more ‘if’ statement (cf. Table 4.1). Also the program length LOC prefers the first implementation, because it is shorter. The value of dep-degree witnesses that the dependency structure of the second implementation is less complicated (dep-degree = 8) compared to the first (dep-degree = 11). The halstead’s indicators also suggest that the first implementation is more ‘difficult’ and requires more ‘mental effort’ to be developed and maintained.

The use-def graphs for both implementations of the ‘equals’ method are shown in Figures 4.6 and 4.7.

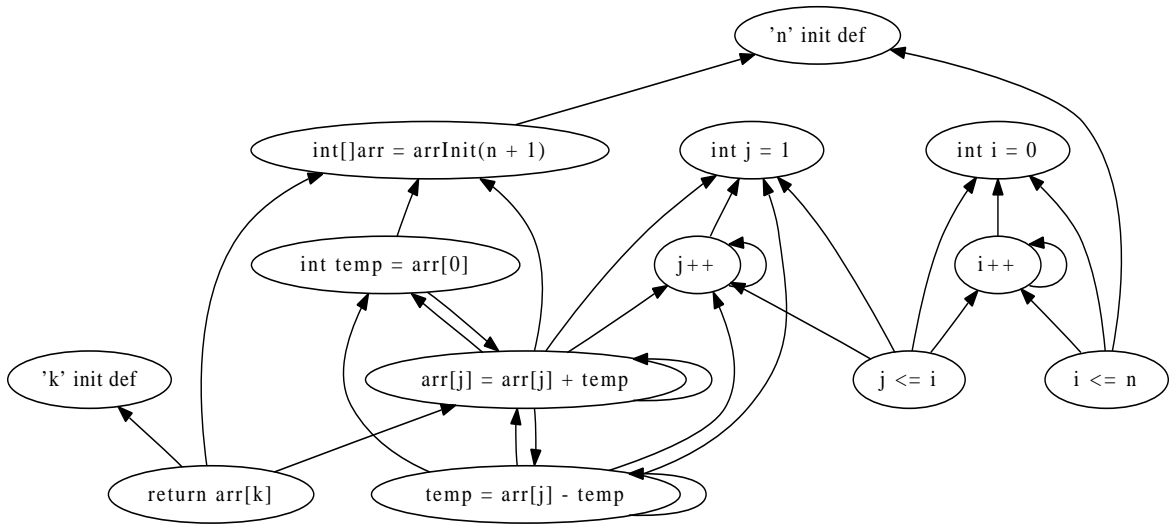


Figure 4.4: Use-def graph for Figure 4.3 (left)

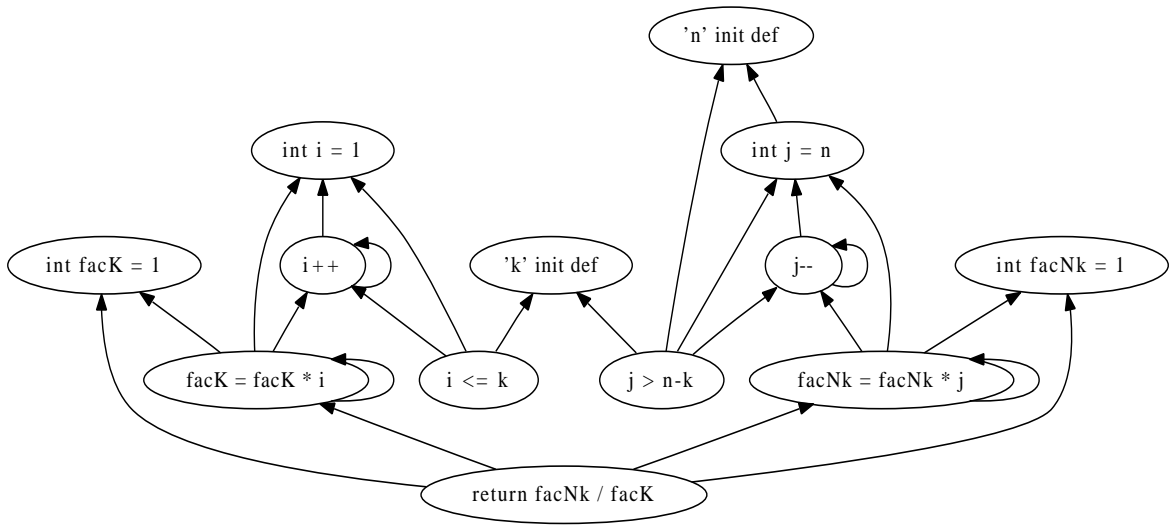


Figure 4.5: Use-def graph for Figure 4.3 (right)

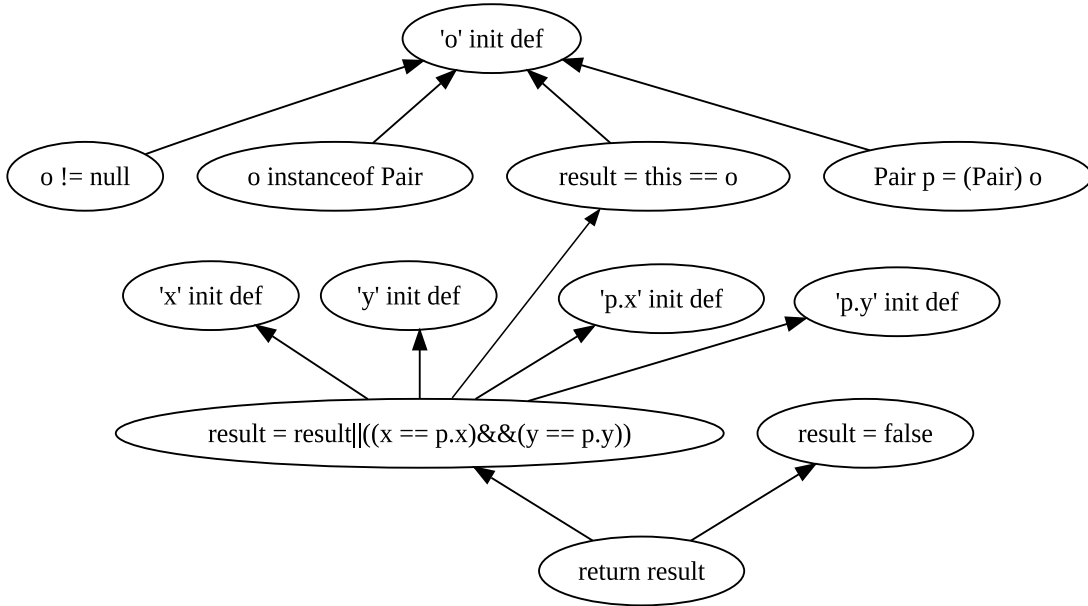


Figure 4.6: Use-def graph for Figure 4.8 (left)

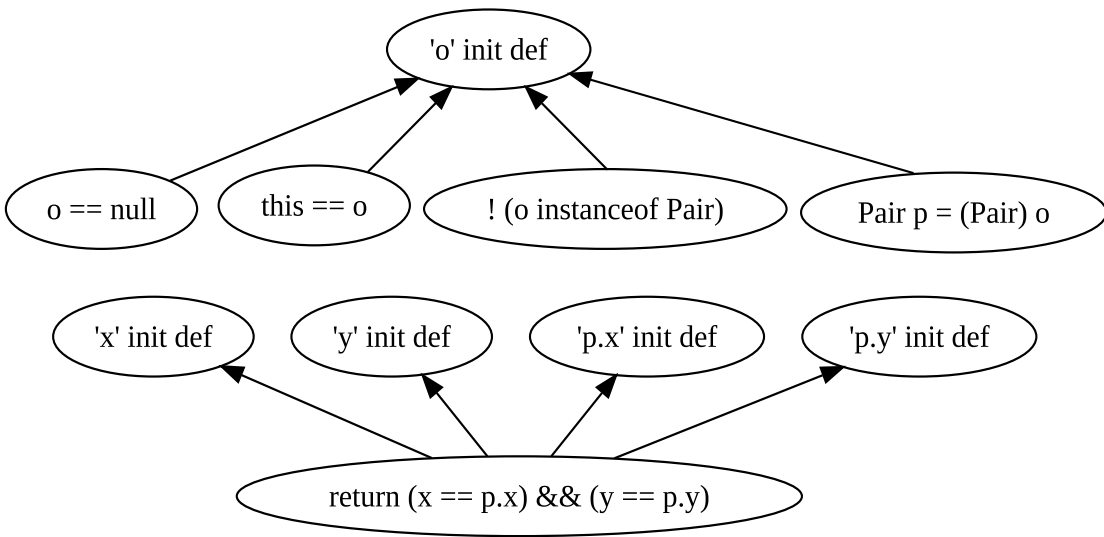


Figure 4.7: Use-def graph for Figure 4.8 (right)

```
class Pair {
  int x;
  int y;
  boolean equals(Object o) {
    boolean result = false;
    if (o != null) {
      if (o instanceof Pair) {
        result = this == o;
        Pair p = (Pair) o;
        result = result ||
          ( x == p.x) && (y == p.y );
      }
    }
    return result;
  }
}

class Pair {
  int x;
  int y;
  boolean equals(Object o) {
    if (o == null) {
      return false;
    }
    if (this == o) {
      return true;
    }
    if (! (o instanceof Pair) ) {
      return false;
    }
    Pair p = (Pair) o;
    return (x == p.x) && (y == p.y);
  }
}
```

Figure 4.8: Two ‘equals’ implementations

Chapter 5

Applications of Dep-Degree

In this chapter we illustrate the effectiveness of our simple indicator for assessing code changes (refactorings) and for indicating complex dependency structure.

5.1 Assessment before and after Refactoring

Our goal in this section is to show that dep-degree indicates structural improvements. Therefore, we explore several concrete code examples that we already know are considered good refactorings (we use them as ‘authoritative’ examples) and test if the indicator agrees. We first discuss a few classic examples of refactoring from (or based on the techniques present in) Fowler’s book [9]. Then we extract several examples from an open-source software project. We selected code commits in which the programmers claim (through the commit logs) that the change was a refactoring to improve the code structure. We are interested in exploring if our indicator dep-degree agrees.

There is no other simple indicator for structural improvement available, and therefore, as done in the last section, we compare dep-degree with the widely used indicators lines of code, cyclomatic complexity and two other well-known indicators introduced by Maurice Halstead, i.e. Halstead’s difficulty and effort indicators [12]. Lines of code (LOC) measures the length of code; cyclomatic complexity (CC) measures the difference of control-flow nodes and edges; Halstead’s difficulty and effort indicators are based on some preliminary measures already mentioned in section 2.4. There were many attempts to define measures for code complexity, but none of them is extensively used in practice. Dep-degree does not measure code complexity either, but is an indicator for complex dependencies.

Method	Version	LOC	CC	DD	HD	HE
printOwing	Original	20	5	27	12.5	6779.2
	Refactored	1	1	1	1.0	20.7
getOutstanding	Original	0	0	0	0.0	0.0
	Refactored	7	2	8	9.0	1188.0
getTaxRate	Original	0	0	0	0.0	0.0
	Refactored	6	4	1	2.5	212.8
printDetails	Original	0	0	0	0.0	0.0
	Refactored	6	1	6	3.4	453.1
log	Original	0	0	0	0.0	0.0
	Refactored	1	1	2	1.0	8.0
<i>TOTAL</i>	Original	20	5	27	12.5	6779.2
	Refactored	21	9	18	16.9	1882.6

Table 5.1: The ‘Extract Method’ example before and after refactoring

5.1.1 Extract Method

The ‘refactoring’ rule that deserves the name refactoring most is the ‘Extract Method’ rule: for a given code fragment, it recommends to *factor out* a cohesive, common, possibly repeating ‘chunk’ of code and move it to a new method (function). We revisited (an extended version of) the method mentioned by Fowler which prints the amount of money a customer owes (`printOwing`). We extracted four new methods: ‘getOutstanding’, ‘getTaxRate’, ‘printDetails’, and ‘log’ (Figure 5.1).

We now wish to check if we actually improved the code. We calculate the values of the three indicators LOC, cyclomatic complexity, and dep-degree, in order to assess the code. For our example of refactoring `printOwing` we know the result already, as Fowler presents good arguments why the refactored code is better. Therefore, we need an indicator that matches this.

Table 5.1 presents the indicator values for lines of code (LOC), cyclomatic complexity (CC), dependency degree (DD), and Halstead’s difficulty (HD) and effort (HE). The value 0 indicates that the method was empty before the refactoring, i.e., did not exist. Only the new indicator DD and Halstead’s effort (HE) correctly identify the improvement of the code: The DD value for `printOwing` was 27 before, and the sum over all new methods is 18; the

the total HE value for the refactored version is also smaller than the value for the original version of `printOwing`; the other two indicators suggest that the new code is longer (LOC increased by one from 20 to 21), has a more complicated control flow (CC increased by four from 5 to 9), and is more difficult overall (HD increased from 12.5 to 16.9).

5.1.2 Inline Method

Sometimes there might be too much ‘indirection’ in the code. For example some methods (functions) might do simple delegation to other methods, and this can cause confusion or make the code less readable because it becomes easier to get lost while tracking all the delegation.

‘Inline method’ is a refactoring that deals with this problem by removing simple or redundant methods and putting their bodies into the body of their caller. In a way, it is the reverse of the process in ‘Extract Method’ refactoring.

Another application of this refactoring is when there is a group of badly refactored methods, which can be inlined to one big method, and then possibly be re-extracted in a better form.

Figure 5.2 shows an instance of ‘Inline Method’ refactoring. The ‘append’ method takes an array of strings and a single string object as input, and returns a collection of strings (`ArrayList<String>`). After the execution of the method is complete, the returned collection contains all the string items that belong to the input array with the original order preserved, plus the single string object appended to the end. Repackaging the array into ‘ArrayList’ is delegated to another method called ‘repackage’. However, repackaging could simply be done within the ‘append’ method (as shown in the refactored version) because it is already clear from the signature and the return type of the append method what it does.

Table 5.2 shows that all indicators except HE agree that the refactored version has simplified the code. This example also shows that breaking a method into smaller ones (e.g. using a process as in ‘Extract Method’ refactoring) is not necessarily always a good idea. It requires a good reason to select and extract a chunk of code out of a method and move it to a new method, and care on how to perform this operation. Otherwise, the code becomes complicated with the introduction of redundant indirection and additional dependencies.

```

// Original Version
void printOwing(Province province) {
    Enumeration<Order> e = _orders.elements();
    double outstanding = 0.0;
    double taxRate;

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // tax rates
    switch (province) {
        case ALBERTA: taxRate = 0.05; break;
        case BRITISH: taxRate = 0.12; break;
        case ONTARIO: taxRate = 0.13; break;
        default: taxRate = 0.10; break;
    }

    // print details
    System.out.println("Name: " + _name);
    System.out.println("Amount: $"
        + outstanding);
    System.out.println("Tax: $"
        + outstanding * taxRate);
    System.out.println("-----");
    System.out.println("Total: $"
        + outstanding * (1 + taxRate));
}

// Refactored Version
void printOwing(Province province) {
    printDetails(getOutstanding(),
        getTaxRate(province));
}

double getOutstanding() {
    double result = 0.0;
    Enumeration<Order> e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}

double getTaxRate(Province province) {
    switch (province) {
        case ALBERTA: return 0.05;
        case BRITISH: return 0.12;
        case ONTARIO: return 0.13;
        default: return 0.10;
    }
}

void printDetails(double outstanding,
    double taxRate) {
    log("Name: " + _name);
    log("Amount: $"
        + outstanding);
    log("Tax: $"
        + outstanding * taxRate);
    log("-----");
    log("Total: $"
        + outstanding * (1 + taxRate));
}

void log (String message) {
    System.out.println(message);
}

```

Figure 5.1: An Example of ‘Extract Method’ Refactoring

Method	Version	LOC	CC	DD	HD	HE
append	Original	4	1	4	3.3	184.5
	Refactored	8	2	12	11.8	1720.1
repackage	Original	7	2	10	11.9	1510.0
	Refactored	0	0	0	0.0	0.0
<i>TOTAL</i>	Original	11	3	14	15.2	1694.5
	Refactored	8	2	12	11.8	1720.1

Table 5.2: The ‘Inline Method’ example before and after refactoring

5.1.3 Introduce Parameter Object

In many cases a group of parameters tend to be passed (to functions) together. This might be due to a natural relation between these parameters, or simply because several functions require all these parameters to perform their tasks. In any case, such situations suggest the idea of consolidating these parameters into a solid entity (e.g. an object in the context of object-oriented languages).

The idea represents a refactoring technique which has certain benefits. One advantage is that it reduces the size of the parameter lists and consequently the difficulty in understanding a code, since long parameter lists are harder to understand. Another (deeper) benefit is that once the parameters are bundled together, one would likely notice common manipulations of the parameter values in the bodies of functions which can be refactored as a behavior and moved into the new object.

The small example shown in Figure 5.3 illustrates these ideas. However, one has to keep in mind that the effect of this refactoring and the difference between indicator values would likely be much more significant for larger and more complicated examples such as those in the Subsection 5.1.7.

Both the original and refactored code use a ‘painter’ object to draw a line using the ‘Graphics’ type in java ‘awt’ package. The original version simply passes the coordinates of the line’s end points as well as its width to the painter, whereas the refactored version uses a parameter object of type ‘Edge’ to transfer the same information to the painter. It is apparent how the reduced parameter list in the refactored version has made the code simpler and more readable. It has decreased the dep-degree value (Table 5.3) because the

```

// Original Version

ArrayList<String>
  append(String[] col,
         String newItem)
{
  ArrayList<String> result =
    repackage(col);

  result.add(newItem);
  return result;
}

ArrayList<String>
  repackage(String[] col)
{
  ArrayList<String> result
    = new ArrayList<String>();
  for (int i=0; i < col.length; i++)
  {
    result.add(col[i]);
  }
  return result;
}

// Refactored Version

ArrayList<String>
  append(String[] col,
         String newItem)
{
  ArrayList<String> result =
    new ArrayList<String>();
  for (int i = 0; i < col.length; i++)
  {
    result.add(col[i]);
  }
  result.add(newItem);
  return result;
}

```

Figure 5.2: An Example of ‘Inline Method’ Refactoring

refactored version uses less variables, i.e. the coordinate and width parameters are replaced by an Edge object. However, the values of LOC and CC indicators remain unchanged. The value of HD is slightly decreased. HE is also decreased from 740.4 to 474.3.

Assuming that switching the x and y coordinates in the above example is common, as suggested by the ‘swapAxes’ flag (e.g. to calculate the reflection against the diagonal axis $y = x$), this behavior has been represented by a method called ‘swapXY’ within the new type ‘Edge’.

Method	Version	LOC	CC	DD	HD	HE
putEdge	Original	6	2	14	6.6	740.4
	Refactored	6	2	7	6.3	474.3

Table 5.3: The ‘Parameter Object’ example before and after refactoring

<pre>// Original Version void putEdge(Graphics g, float width, int xs, int ys, int xt, int yt, boolean swapAxes) { OriginalPainter painter = new OriginalPainter(); if (!swapAxes) painter.drawEdge(g, width, xs, xt, ys, yt); else painter.drawEdge(g, width, ys, yt, xs, xt); } </pre>	<pre>// Refactored Version void putEdge(Graphics g, Edge e, boolean swapAxes) { RefactoredPainter painter = new RefactoredPainter(); if (!swapAxes) painter.drawEdge(g, e); else painter.drawEdge(g, e.swapXY()); } </pre>
--	--

Figure 5.3: An Example of ‘Introduce Parameter Object’ Refactoring

5.1.4 Parameterize Method

Sometimes it is possible to remove duplicate code by replacing the repetitive or similar pieces of code with a single method (function) that handles the variations by parameters. This also increases the flexibility because it makes it possible to deal with new variations simply by adding parameters. For example, there might be several methods that do similar things but with different values hardcoded in the method body. One can replace these similar methods by a single one that uses a parameter for different values.

The example in Figure 5.4 is an extended version of the one by Fowler [9] which is an instance of this refactoring. The dep-degree value of the refactored version is less than that of the original one due to the removal of repetitive code and also the dependencies caused by conditional statements (Table 5.4). Therefore the dep-degree indicator shows that the refactoring has improved the code. The two indicators LOC and cyclomatic complexity indicate a small improvement in the refactored version as they are both decreased by one unit. Both HD and HE also show improvement in the code.

5.1.5 Pull Up Method

Some methods might have similar or identical purposes within subclasses in a class hierarchy. They might simply return the same results, or even share the same body. This usually suggests that these methods might be too abstract to belong to the their owner classes, and that they should be moved to the superclass.

Method	Version	LOC	CC	DD	HD	HE
baseCharge	Original	17	4	13	15.4	4599.6
	Refactored	9	1	5	6.7	1114.5
usageInRange	Original	0	0	0	0.0	0.0
	Refactored	7	2	4	4.9	316.2
<i>TOTAL</i>	Original	17	4	13	15.4	4599.6
	Refactored	16	3	9	11.6	1430.7

Table 5.4: The ‘Parameterize Method’ example before and after refactoring

This refactoring is called ‘Pull Up Method’. It usually improves the code structure by removing duplicate code. Even though two duplicate methods might work fine the way they are, but there is the risk that one might be updated when the other is left forgotten. Also, the code becomes less readable when crowded with duplicate code.

Figure 5.5 shows the effect of this refactoring on code simplification. The employees of a company are modeled using a type hierarchy system. It is assumed that each employee is either an engineer or a salesperson. Each of these job titles is associated with a class named after it, and these classes are subclassed from a more general type called ‘Employee’. The Employee interface is a way to handle general references and inquiries to the employee objects representing the employees of the company. For example, both ‘printInfo’ methods in Figure 5.5 take an object containing the records of an employee and print his/her name and work experience. However, in the left example (original version), ‘Employee’ is just a Java interface containing only the signatures of subclasses’ methods, whereas in the right one (refactored version) it is an abstract class. This means that the employee data fields (such as name) and related getter methods (e.g. getName) can be pulled up to the Employee abstract class and removed from the subclasses. This makes sense since each employee has a name and a work experience record regardless of his/her job title. It can be observed how this refactoring affects the way a general inquiry (such as asking for an employee name) takes place in the code. As shown in Table 5.5, all the indicators confirm that the printInfo method that uses refactored types is simpler and more understandable than the other one, because the refactored types remove the need for type matching, as a result of which the if statements and the related dependencies are omitted.

<pre>// Original Version double baseCharge() { double result = 0.03 * Math.min(lastUsage(),100); if (lastUsage() > 100) { result += 0.05 * (Math.min(lastUsage(),200) - 100); } if (lastUsage() > 200) { result += 0.07 * (Math.min(lastUsage(),300) - 200); } if (lastUsage() > 300) { result += (lastUsage() - 300) * 0.09; } return result; }</pre>	<pre>// Refactored Version double baseCharge() { double result = 0.03 * usageInRange(0, 100); result += 0.05 * usageInRange(100, 200); result += 0.07 * usageInRange(200, 300); result += 0.09 * usageInRange(300, Integer.MAX_VALUE); return result; } int usageInRange(int start, int end) { if (lastUsage() > start) { return Math.min(lastUsage(), end) - start; } else return 0; }</pre>
--	--

Figure 5.4: An Example of ‘Parameterize Method’ Refactoring

5.1.6 Replace Conditional with Polymorphism

Polymorphism is a feature of object-oriented programming which removes the need to include explicit conditional statements when there are objects whose behavior varies based upon their type. When there is a conditional statement that decides which behavior to select depending on the type of an object, it is possible to avoid such statement by “moving each leg of the conditional to an overriding method in a subclass” [9]. Figure 5.6 shows such an example. In the original version, constants (shown in capital letters) are used to represent different employee types. However, in the refactored version actual types are used to denote different employee titles (e.g. engineer, salesperson), each of which is a subtype of the abstract class ‘Employee’. The method ‘payAmount’, which calculates the salary of

Method	Version	LOC	CC	DD	HD	HE
printInfo	Original	20	3	15	11.3	3211.9
	Refactored	5	1	5	3.6	281.7

Table 5.5: The ‘Pull Up Method’ example before and after refactoring

Method	Version	LOC	CC	DD	HD	HE
payAmount	Original	17	4	10	9.6	2012.1
	Engineer	2	1	2	1.5	17.4
	Manager	2	1	3	1.5	27.1
	Salesperson	2	1	3	1.5	27.1
<i>TOTAL</i>	Original	17	4	10	9.6	2012.1
	Refactored	6	3	8	4.5	71.6

Table 5.6: The ‘Replace Conditional with Polymorphism’ example before and after refactoring

each employee based on his/her title (object type), is overridden with alternative implementations for each subtype. Polymorphism is used to call the appropriate ‘payAmount’ of an Employee reference based on its underlying type. Table 5.6 lists the indicator values for the given example.

5.1.7 Revisions from Code Repositories

In the following we assess several series of refactorings made to a small-sized software project (CCVISU). The structural improvements resulted from a variety of refactoring schemas, including ‘Extract Class’, ‘Extract Method’, ‘Move Method’, ‘Move Field’, ‘Parameter Object’, and ‘Replace Array with Object’. We refer to the different software versions via SVN revision numbers. The revisions are conveniently accessible using the following URL:

<http://code.google.com/p/ccvisu/source/detail?r=44>

where the number 44 is the SVN revision number.

ParseCmdLine. First we consider some refactorings that took place between revisions 28 and 34, i.e. a set of refactorings that were applied step by step to the method `main` of the main class: a new type `Options` was extracted, an instance of which serves as a

<pre>// Using Original Types void printInfo(Employee emp) { String name = null; int workYears = 0; if (emp instanceof Engineer) { Engineer engineer = (Engineer) emp; name = engineer.getName(); workYears = engineer.getWorkExperience(); } else if (emp instanceof Salesperson) { Salesperson salesPerson = (Salesperson) emp; name = salesPerson.getName(); workYears = salesPerson.getWorkExperience(); } System.out.println("Name: " + name + " - Work Experience (in years): " + workYears); } </pre>	<pre>// Using Refactored Types void printInfo(Employee emp) { String name = emp.getName(); int workYears = emp.getWorkExperience(); System.out.println("Name: " + name + " - Work Experience (in years): " + workYears); } </pre>
--	--

Figure 5.5: An Example of ‘Pull Up Method’ Refactoring

‘parameter object’ to many constructor calls, and a method `parseCmdLine` for parsing the command-line arguments was extracted from the method `main` and moved to the new type. Table 5.7 shows the indicator values for the method `main` of revision 28 and for the methods `main` and `parseCmdLine` of revision 34.

LOC suggests that the refactorings have slightly improved the code, the cyclomatic complexity suggests that the code became slightly worse after refactoring, and dep-degree suggests that the code *significantly* improved. Halstead’s difficulty indicates that the refactoring has made the code more difficult, and Halstead’s effort suggests the code has been improved by the refactoring.

Enums instead of ‘final int’. Revision 20 of CCVISU refactors some code in order to introduce ‘enum’ (enumeration) types. In older versions of Java, developers had to ‘simulate’ enums by several final integer (‘int’) members, as in the non-refactored code in Figure 5.6. Since recent versions, Java supports enums natively, and revision 21 transformed the code accordingly. Table 5.8 shows the effect of this refactoring on the indicator values

<pre>// Original Version public class Employee { static final int ENGINEER = 0; static final int SALESMAN = 1; static final int MANAGER = 2; // ... int payAmount(int overtimePay) { int basePayment = _monthlySalary; switch (getType()) { case ENGINEER: basePayment += _experience; break; case SALESMAN: basePayment += _commission; break; case MANAGER: basePayment += _bonus; break; default: throw new RuntimeException("Incorrect Employee"); } return basePayment + overtimePay; } }</pre>	<pre>// Refactored Version abstract class Employee { protected int _monthlySalary; // ... abstract int payAmount(int overtimePay); } class Engineer extends Employee { @Override int payAmount(int overtimePay) { return _monthlySalary * _experience + overtimePay; } } class Manager extends Employee { @Override int payAmount(int overtimePay) { return _monthlySalary + _bonus + overtimePay; } } class Salesperson extends Employee { @Override int payAmount(int overtimePay) { return _monthlySalary + _commission + overtimePay; } }</pre>
--	---

Figure 5.6: An Example of ‘Replace Conditional with Polymorphism’ Refactoring

for the methods `getInFormat` and `getOutFormat`. All the indicators agree: this change is a significant improvement.

Colors. A new type `Colors` was introduced in revision 38 to better organize the access of colors and to help simplify the code. Table 5.9 shows the values of the indicators for the participating methods of this refactoring: all indicators agree that the new version is better, except for one method where the cyclomatic complexity does not change, and two methods for which the indicators HD and HE remain unchanged after refactoring.

ParseCmdLine refactored. The method `parseCmdLine` was further significantly simplified by refactoring a data structure that was frequently used. It decreased the number

Method	Revision	LOC	CC	DD	HD	HE
CCVisu.main	r28	328	77	512	56.6	554288.2
	r34	81	27	64	38.0	78786.1
Options.parseCmdLine	r28	0	0	0	0.0	0.0
	r34	215	53	331	32.9	194362.5
<i>TOTAL</i>	r28	328	77	512	56.6	554288.2
	r34	296	80	395	70.9	273148.6

Table 5.7: Refactoring ‘parseCmdLine’ from r28 to r34

Method	Rev	LOC	CC	DD	HD	HE
getInFormat	r20	15	5	17	9.5	2449.9
	r21	9	2	7	5.8	723.3
getOutFormat	r20	17	6	20	10.0	3068.6
	r21	9	2	7	5.8	723.3
<i>TOTAL</i>	r20	32	11	37	19.5	5518.5
	r21	18	4	14	11.6	1446.6

Table 5.8: Enum change from r20 to r21

Method	Rev	LOC	CC	DD	HD	HE
ClusterManager.ClusterManager	r37	261	1	186	10.3	30147.9
	r38	225	1	168	10.3	30147.9
ClusterManager.refreshInfo	r37	47	20	70	12.5	16000.4
	r38	12	2	19	7.5	2425.0
Options.parseCmdLine	r37	173	47	296	29.1	135576.5
	r38	164	43	284	28.9	123946.5
ScreenDisplay.ScreenDisplay	r37	495	31	326	15.1	72772.2
	r38	456	13	307	15.1	72772.2
<i>TOTAL</i>	r37	976	99	878	67.0	254497.0
	r38	857	59	778	61.8	229291.6

Table 5.9: Better color handling from r37 to r38

Method	Rev	LOC	CC	DD	HD	HE
Options.parseCmdLine	r43	162	43	282	29.0	123164.9
	r44	162	43	154	21.2	92120.2
	r117	172	45	158	24.3	112473.4
	r118	155	45	125	21.6	89918.4

Table 5.10: Improvements from r43 to r44 and from r117 to r118

of variables used in each statement, and encapsulated some unnecessary details behind the interface of the new data structure. This refactoring took place in revision 44. Another refactoring was applied to the same method in revision 118 to further improve the method: method `chkAvail` was extended by a new feature and got renamed to `getNext`. Table 5.10 reports that cyclomatic complexity does not notice these changes (value unchanged); LOC stays unchanged for the refactoring at revision 44. Dep-degree confirms a significant improvement in the dependency structure. Halstead's difficulty suggests that the code has become slightly more difficult in revision 118 as compared with revision 44, whereas dep-degree indicates that it has become simpler.

5.1.8 Position objects instead of 3-dim arrays

Another major refactoring was performed step by step throughout the revisions 112 to r117. Positions in the 3-dimensional space were represented by a 3-dimensional array. During the refactoring, the code was transformed in order to use objects of a class `Position`, which encapsulates the three coordinates, and provides common operations of positions as vector operations. Many disturbing lines of code were removed from several methods, one method got removed. More specifically, the code that was implementing operations that were truly vector operations, got explicitly moved to new methods. Table 5.11 reports that LOC slightly decreased in total. The value of cyclomatic complexity is slightly higher, i.e., would not recommend the refactoring. The value of dep-degree confirms this change as a structure-improving refactoring. Halstead's difficulty and effort indicators also suggest that the refactoring has simplified the code. (The value 0 in the table indicates an empty method body, again.)

5.1.9 Summary of Results

Throughout the section 5.1, the new indicator dep-degree, along with four other widely used or well-known indicators (namely lines of code, cyclomatic complexity, and Halstead's difficulty and effort) were evaluated on several examples of refactoring. Some examples are extended versions of the sample refactorings presented in the refactoring book by Fowler [9]; some others are inspired by the refactoring methods he discusses; and the rest of the examples are the application of refactorings to the source code of an open-source project, CCVisu. Each example consists of an initial implementation and a refactored version of it.

The results show that the dep-degree indicator satisfies the expectations and confirms that in all the given examples the refactorings have improved the dependency structure of the code. The other indicators sometimes 'agree' with the refactoring, but other times either find the refactoring neutral or vote against it occasionally. Therefore, in comparison to the indicators lines of code, cyclomatic complexity, and Halstead's difficulty and effort which are often used for measuring maintainability and understandability, dep-degree is a relatively better indicator for assessing code improvements between different versions, e.g., by refactoring.

5.1.10 Possible Limitations

There might be some limitations to the newly proposed indicator, dep-degree. Due to the fact that dep-degree is based on the low-level software dependencies (between program operations), it might not be able to detect certain high-level design improvements, particularly the changes that hardly affect the code-level dependencies.

Another possible limitation is when there is no variable involved in the program code. In such cases, the dep-degree value would be zero, since dep-degree measures the number of reaching definitions for the variables used by each program operation.

Another example where dep-degree might not truly reflect the understandability of the code is a program in which a variable can be initialized to multiple values depending on the value of a parameter (e.g. when a switch statement with numerous branches is used to initialize a variable). In this case, any successor operation that uses the initial value of the variable is assigned a high indicator value, due to the large number of possibilities involved. Nonetheless, it is likely that a developer realizes the initialization process quite easily by a quick look at the program code, which is against the prediction of the indicator and its high

value.

5.2 Indication of Problematic Code

In the last subsection we applied our indicator to the assessment of code changes as they occur during refactorings, and showed that the dep-degree values match the developer opinion, and that LOC and cyclomatic complexity are not applicable to assessing refactorings. We would now like to point out two possible applications for using dep-degree for indicating problematic code, i.e., to *indicate* and *locate* the pieces of code with the most complex dependencies (which then could be considered for refactoring). The adoption of measures for refactoring inference has been found to be useful (e.g., [6, 33]), and we believe that our indicator can be used to complement those approaches.

5.2.1 Identifying problematic operations

Dep-degree for program operations can be applied to single program operations. We implemented a tool that generates a detailed report of the dep-degree values for each operation in the program. This immediately proposes two possible uses: (a) we can list and inspect the operations with highest dep-degree values, and (b) we can color (highlight) each operation in the source code editor according to its dep-degree value.

We have implemented (a) and (b) in an Eclipse plug-in. For (a), our plug-in tool attaches markers to the left vertical ruler of Eclipse’s main editor. These markers locate operations with highest dep-degree values in the selected source file. Figure 5.7 shows a snapshot of the tool where a yellow window has popped up listing the markers on the left vertical ruler of the editor. The markers summarize the operations with highest dep-degree values. For (b), we generate a relative color map *rgb* that assigns to each dep-degree value a color on the scale from white to red, where 0 is mapped to white with $rgb(0) = (255, 255, 255)$ and a predefined maximum dep-degree value (dd_{max}) is mapped to red with $rgb(dd_{max}) = (255, 0, 0)$. dd_{max} can be set for example to 9 ($= 7 + 2$), so that any operation with the dep-degree value of over 8 is highlighted as ‘too complicated’ (the number 9 is inspired by Miller’s article regarding the limit on our capacity to process information [26]). Figure 5.8 shows a snapshot of the tool where program operations are colored according to their dep-degree values. Operations with higher values are highlighted with a darker red color.

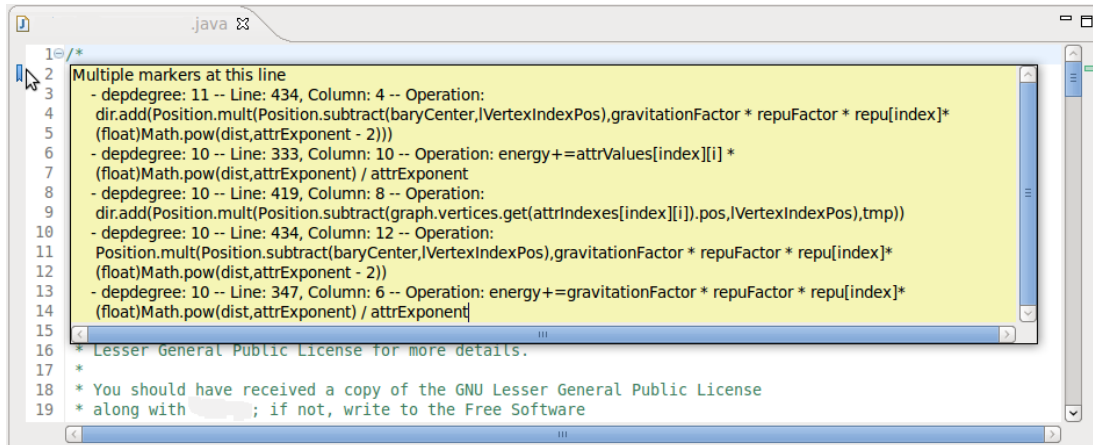


Figure 5.7: Markers locating operations with highest dep-degree values

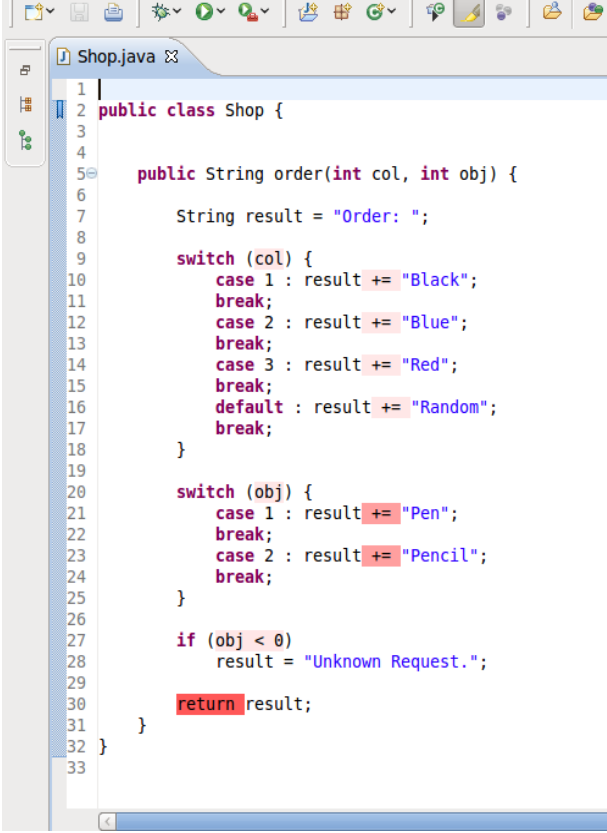
Also, if an operation is selected in the editor, then all defining operations (reaching definitions) for the selected operation are highlighted.

5.2.2 Identifying problematic functions

Dep-degree for program functions can be applied to yield a value for each program function. The application ideas for the operation level can be ‘lifted’ to function level, i.e., we can generate the dep-degree values for program functions and (a) look for outliers in the sorted/ranked list of program functions and annotate functions with their dep-degree values, or (b) assign colors to the graphical objects that represent functions in software visualizations.

Method	Rev	LOC	CC	DD	HD	HE
minimizeEnergy	r112	91	15	125	62.7	185610.9
	r117	86	15	94	45.4	105963.5
getDirection	r112	44	6	86	43.7	62779.2
	r117	34	4	70	35.0	37983.2
addRepulsionDir	r112	20	7	29	20.9	13464.1
	r117	19	6	25	16.3	8953.3
computeBaryCenter	r112	11	1	12	10.7	2167.0
	r117	5	1	4	4.2	247.0
getDist	r112	4	1	10	9.8	1758.5
	r117	3	1	7	8.4	752.8
getDistToBaryCenter	r112	4	1	10	9.8	1758.5
	r117	0	0	0	0.0	0.0
Position (Class)	r112	0	0	0	0.0	0.0
	r117	26	6	56	31.5	2238.9
<i>TOTAL</i>	r112	174	31	272	157.6	265779.7
	r117	173	33	256	140.8	156138.7

Table 5.11: Improved Position from r112 to r117



```
1 |
2 | public class Shop {
3 |
4 |
5 | public String order(int col, int obj) {
6 |
7 |     String result = "Order: ";
8 |
9 |     switch (col) {
10 |         case 1 : result += "Black";
11 |         break;
12 |         case 2 : result += "Blue";
13 |         break;
14 |         case 3 : result += "Red";
15 |         break;
16 |         default : result += "Random";
17 |         break;
18 |     }
19 |
20 |     switch (obj) {
21 |         case 1 : result += "Pen";
22 |         break;
23 |         case 2 : result += "Pencil";
24 |         break;
25 |     }
26 |
27 |     if (obj < 0)
28 |         result = "Unknown Request.";
29 |
30 |     return result;
31 | }
32 |
33 |
```

Figure 5.8: Coloring program operations according to their dep-degree values

Chapter 6

Tool Implementation

In this chapter we present a software tool that we implemented to support the evaluation of our indicator [4]. We also discuss the complexity of the algorithm we developed for the implementation.

6.1 Algorithm

It is usually easy to manually calculate the values of the dep-degree indicator for small or even some medium-sized programs. However, for larger and more complicated programs this might not be feasible.

We have developed a software tool to automate the calculation of the proposed indicator for Java programs. We implemented a simple and efficient algorithm to compute the reaching definitions. However, one should note that only an approximate calculation of the set of the reaching definitions is possible. The precise calculation of the reaching definitions requires alias analysis, a technique in compiler theory that is used to determine whether a memory location is referred to by different variables (or pointers). But the problem of statically determining alias information has been proved to be undecidable [22, 32].

The CFG of a program can be obtained through its abstract syntax tree which is created while parsing the source code. In order to evaluate the indicator value, one has to calculate for each program operation the size of the set of the reaching definitions of that operation. For each variable used by an operation, the CFG can be traversed backward using a procedure based on the depth-first search (DFS) algorithm, starting from the node representing the operation. While running the ‘backward’ DFS, the graph traversal of the path currently

being visited halts as soon as a reaching definition with respect to the variable is discovered. The DFS itself continues until all the accessible paths are explored.

Algorithm 1 Computing the reaching definitions

```

for each Program Operation in the Function do
  for each Variable Used by the Program Operation do
    Run A Backward DFS on the Control-Flow Graph
  end for
end for

```

Consider a program function and let the directed graph G and the integers s and o be:

$G = (V, E)$: The control-flow graph of the function with the set of vertices V and the set of edges E

s : The total number of program operations within the function

o : The maximum number of variable occurrences in any program operation

The (backward) DFS algorithm takes $O(|V| + |E|)$ every time it is invoked. Therefore the complexity of the algorithm which calculates the dep-degree value for the function would be $O(so(|V| + |E|))$.

We can assume – at least for structured code – that the difference between the number of vertices and edges in a CFG is negligible. For example a method with a cyclomatic complexity of over 10 is considered too complicated [25]. This means:

$$cc = |E| - |V| + 2 \leq 10 \implies |E| - |V| \leq 8 \quad (6.1)$$

Therefore the complexity of the algorithm becomes $O(so|V|)$.

6.2 Features and Architecture

Our implemented tool integrates with the Eclipse IDE in the form of a plug-in, and is capable of interacting with the main source code editor of the development environment.

The plug-in uses the Eclipse Java Development Tools (JDT) to parse the selected source file within the active editor and create the abstract syntax tree. Then, the control flow graph of the program is produced by traversing the syntax tree using the available visitor class (as in Visitor pattern) provided by JDT. The set of reaching definitions are computed

for each node in the CFG and the indicator value is calculated for all program operations accordingly.

The tool features a nice GUI and highlights atomic operations of a program in its source text within the editor based on their indicator values. The higher dep-degree value an operation has, the darker is the red color that highlights that operation. It is also possible to view the exact indicator value for each operation in a dedicated text field. By clicking or selecting a program statement in the editor, the set of all the operations representing the reaching definitions of the selected statement are highlighted. The tool also generates a text result which contains the indicator value for each method in the class file and each statement within each method along with the set of reaching definitions of that statement.

Part II

Dependency Analysis on Architecture Level

Chapter 7

Overview

Many software developers use a syntactical ‘diff’ in order to perform a quick review before committing changes to the repository. Other developers are notified of the change by e-mail (containing diffs or change logs), and they review the received information to determine if their work is affected. We lift this simple process from the code level to the more abstract level of dependencies: a software developer can use our software tool (called CHECKDEP) to inspect introduced and removed dependencies before committing new versions, and other developers receive summaries of the changed dependencies via e-mail. We find the tool useful in our software-development activities and would like to make the tool available to others. Our tool does not claim any novel concept, and is considered a purely practical contribution. The results of this study are published in a corresponding conference paper [3].

The objective of CHECKDEP is to provide feedback on the consequences of code changes in terms of dependencies. Our project was motivated by a concrete problem in an industrial project: controller software was continuously extended for new product versions, and the originally well-designed software was degenerating. The problem for maintenance in this project is the amount of inter-dependencies between the various subsystems. A first measure to approach the problem is to make sure that no new dependencies are introduced, and therefore all new changes (commits) are inspected for dependency reduction. CHECKDEP provides a list of changes in the dependency relations for any two versions of the project. A second measure is to continuously refactor the system – whenever the product cycle allows – in order to stepwise transform the system to a better subsystem structure, and to evaluate every refactoring step with respect to the amount of dependencies introduced and removed. CHECKDEP compares the workspace copy with the head revision before check-in,

and notifies the developer of introduced dependencies. A change that reduces the number of inter-subsystem dependencies is considered good under this measure, and a change that introduces more new dependencies than it removes is considered suspicious and the developer is alarmed. In addition to immediate textual feedback, we provide a clustering visualization in order to locate and investigate the dependency changes.

CHECKDEP can be used as a command-line tool or as Eclipse plug-in. Command-line invocations are necessary in automatic processes, such as being called from a Subversion hook script automatically after each commit. The Eclipse plug-in requires the user to work in two steps: First, the user has to specify the dependency types to be extracted (combination of call, inheritance, and field access) and the two versions to compare (either by paths of working directories, or by URLs of Subversion repositories). Second, the processing is done and the textual and visual results are shown. The textual results are a brief summary of the dependency changes and a list of all added and removed dependencies in RSF (Rigi Standard Format). The visualization is based on a clustering layout (CCVISU¹) that easily identifies the area of change in a (large) project graph. CHECKDEP is free software, released under the Apache 2.0 license.

Many existing tools have addressed similar problems (CREOLE², DA4JAVA³, DEPAN⁴, DEPENDENCYFINDER⁵). For example, ‘Creole’ is an Eclipse plugin that integrates ‘Shrimp’ – an application for visualization and exploration of software architecture – into Eclipse platform’s Java Development Tools. ‘D4AJava’ uses nested graphs and special filterings to present static dependencies between software artifacts (elements), easing the task of understanding large dependency structures in the process of incremental composition of graphs. Dependency Finder is a toolset for analyzing compiled Java code, and can be used to extract dependency graphs. Unlike Creole and D4AJava, it does not feature visualization of the dependency graphs, but returns the result in various known formats such as XML, HTML, GraphML and regular text which might be used by other programs (e.g. to visualize the dependencies).

The contribution of CHECKDEP is to provide a small tool for analyzing the change of

¹<http://www.sosy-lab.org/~dbeyer/CCVisu/>

²<http://www.thechiselgroup.com/creole/>

³<http://seal.ifi.uzh.ch/240/>

⁴<http://code.google.com/p/google-depan/>

⁵<http://depfind.sourceforge.net/>

dependencies in different versions (via a smooth integration with Subversion). A unique characteristic of CHECKDEP is its special visualization feature that enables the tool to help navigating large dependency graphs, and allows the user to explore a ‘meaningful’ layout in which related artifacts are displayed closely together.

Chapter 8

Features

Figure 8.1 illustrates the architecture of the CheckDep tool and its components. An arrow in the figure indicates an inquiry from the component located at the head of the arrow.

Here is a list of the main features in CHECKDEP and their descriptions:

Dependency Differences. The goal of CHECKDEP is to provide a ‘diff’ on a more abstract level and therefore it focuses on the difference between the dependencies of different versions.

Subversion Integration. CHECKDEP can be applied to different local workspace copies, but also to remote version repositories (specified by URL and revision number via choice list on the configuration screen).

Visualization. For the visual presentation of the dependency differences, we obtained the best results using a clustering layout. Software artifacts (classes, methods, fields) are drawn as circles which are placed close to each other if they are coupled by many dependencies, and placed at distant positions if they are not coupled by many dependencies. An edge represents the dependency between the two connected artifacts. CHECKDEP highlights new dependencies using red edges and removed dependencies using green edges. This reflects the subsystem structure of the software, and at the same time helps the user find the parts of the system that have changed and navigate through the graph.

Graph Query. In order to better achieve its main goal as a dependency ‘diff’ tool, CHECKDEP provides a simple graph query mechanism that allows a user to focus on the graph changes only. For example the user can have CHECKDEP show a minimal subgraph that contains all the added or removed edges in the most recent view. In other words, if

the user is using the zoom feature to view a subgraph (possibly representing a subsystem), he/she can further restrict the layout to only show a subset of the subgraph that holds all the new or old dependencies. It is possible to view only added or removed vertices (artifacts) as well.

Extraction of Dependency Graphs. CHECKDEP uses a fact extractor that is based on DEPENDENCYFINDER to retrieve relations between methods and fields. We extended the fact extractor to differentiate between call, inheritance, and field dependencies. This enables us to select any of the three dependency types for the analysis.

The fact extractor in CHECKDEP provides accurate and detailed results which allows generation of robust dependency graphs. The reason such accurate dependency analysis is adopted is to avoid ambiguity caused by certain features such as method overloading which could create duplicate entries (vertices) for the same entities (artifacts). This choice, however, is a trade-off with the performance. Nonetheless, our analysis of the tool – while operating on the source code of small and medium-sized software – demonstrates an acceptable performance (Table 8.1). The performance tests are run in a computer with 4 GB of memory and a dual core processor running at 2.10 GHz.

To analyze CHECKDEP’s performance, three open source java projects were chosen as input for the tool, namely CCVISU, CPACHECKER¹ and CHIC4WEB (a web service verification tool). For each project, the source code of two different revisions were compared against each other. Table 8.1 shows for each project the total number of distinct types (classes), the total number of distinct members, the revision number, the times in seconds needed for check-out, build, and fact extraction. The time required for visualizing the results depends on the number of iterations taken by the energy minimizer algorithm. The more iterations are taken by the algorithm, the more accurate layout is calculated. However, the number of iterations can be restricted according to the user’s preference. Usually 100 iterations are enough to obtain a reasonable layout.

¹<http://cpachecker.sosy-lab.org/>

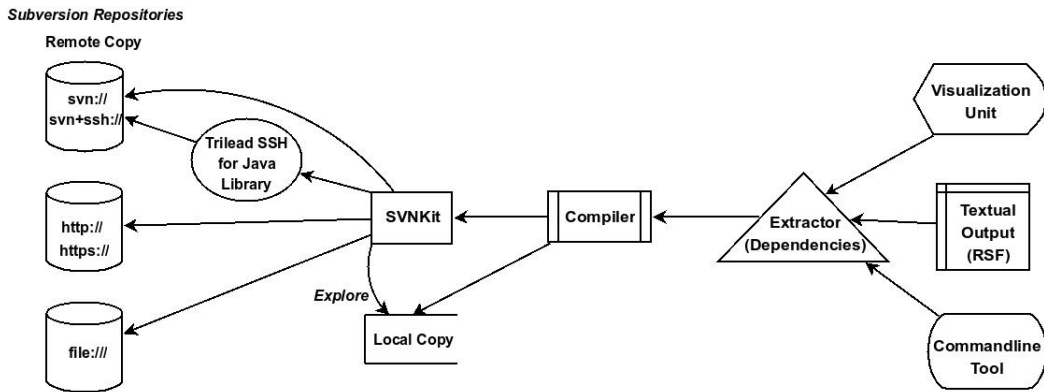


Figure 8.1: CheckDep Architecture

project (types / members)	rev	co	bld	extr
CCVISU (236 / 1296)	28	26.4	6.2	4.8
	34	117	1.4	
CPACHECKER (680 / 4656)	569	35.5	12.4	12.3
	574	33.1	3.0	
CHIC4WEB (133 / 1198)	114	65.2	0.9	2.9
	120	84.8	0.8	

Table 8.1: Tool performance on open-source projects (all times are given in seconds)

Chapter 9

Applications of CheckDep

In this section we list a few useful applications of CHECKDEP.

9.1 Development

The tool can be used to compare the developer's working copy against the head revision of the repository with respect to dependencies, before committing new changes to the repository. The differences in dependencies can be investigated graphically (clustering layout with changed dependencies highlighted) or textually. Filters and zoom-in can be used to restrict the result to a certain part of the software. A search feature can be used to locate specific software elements in the graphical view.

9.2 Refactoring

The clustering layout arranges the software artifacts using well-defined distances, based on their relatedness in the dependency graph. This is useful for inspecting and validating refactoring results, where only a subset of related software artifacts are changed together. The artifacts that participate in a refactoring are related, and therefore, they are closely placed in the layout and easy to locate. The colored edges are highlighting the dependency changes that a refactoring is responsible for. Short edges are not very important, because the artifacts that are placed closely together are already related. The longer an edge is, the more important the dependency is: very long edges represent inter-subsystem dependencies,

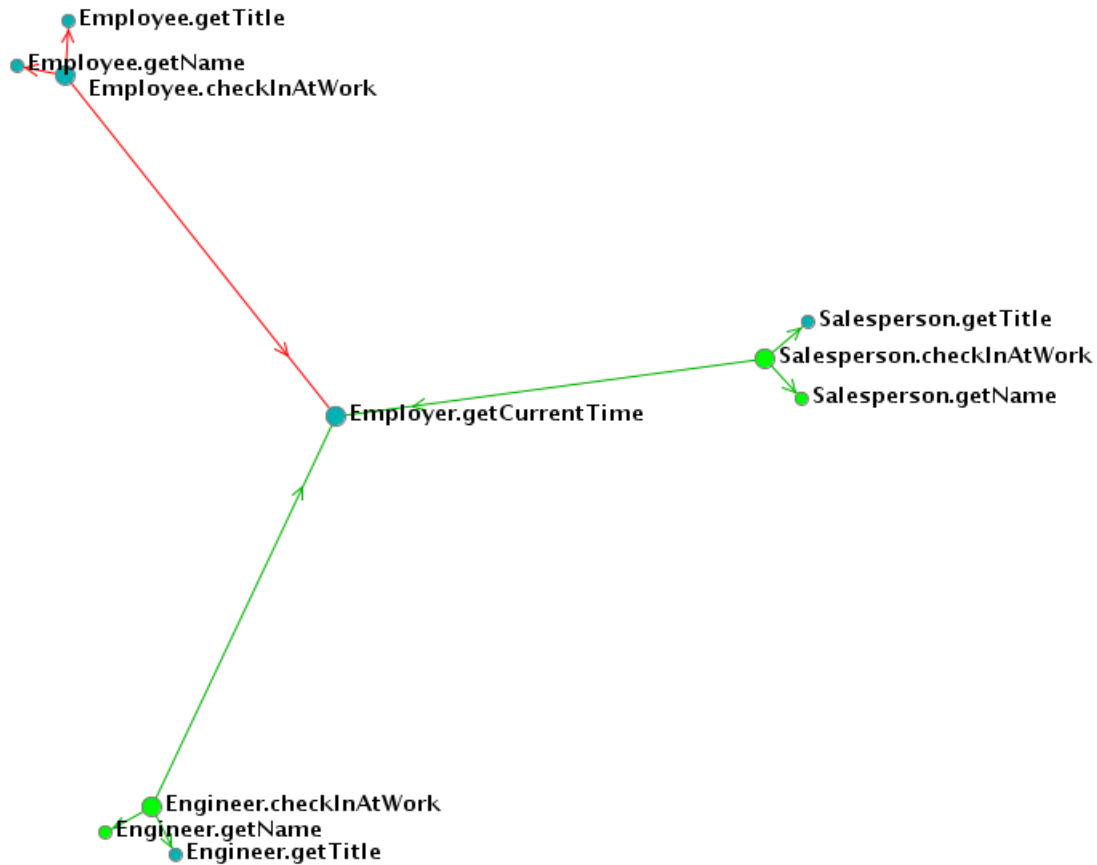


Figure 9.1: A ‘pull-up method’ refactoring removes 6 dependencies (green) and adds 3 (red), which improves the software structure

and removal of such a dependency is a large gain, and introducing such a dependency is degenerating the overall structure of the system in most cases (according to classic definitions, a good structure consists of cohesive subsystems that are loosely coupled). Figure 9.1 shows how CHECKDEP illustrates a local refactoring (lift `getName` and `checkInAtWork` to `Employee`).

9.3 Structure Assessment

In addition to many existing views for type hierarchy and call graphs in Eclipse, CHECKDEP integrates visual clustering (à la CCVISU) into Eclipse. In difference to the available hierarchical or aesthetic layouts, this additional view reflects the relatedness of artifacts by short distances, and separation by long distances. CHECKDEP highlights the changed part

of a system in Fig. 9.2.

9.4 Design Change Identification

Recently an approach was introduced to automatically determine if a change in a source code impacts the design (i.e., UML class diagram) of the related system [13]. Source changes that affect the static design model of a software system (represented by a UML diagram) in a meaningful way are called ‘design changes’. According to the paper, ‘design changes’ are identified in a series of steps, i.e. first by exploring the addition or deletion of classes, then methods, and finally changes in dependency relationships (e.g., generalization, association). This is where CHECKDEP comes in: most of the aforementioned changes are directly identifiable and highlighted within various dependency graphs provided by CHECKDEP. For example added or deleted ‘generalization’ can be realized by added or removed dependencies and nodes in the inheritance graph; also, the added or removed dependencies in the type-field graph denote added or deleted ‘associations’.

Not only is it possible to obtain such information from a simple textual output by CHECKDEP, but the visualization unit also comes in handy for manual, yet convenient examination of changes in the related graphs (e.g. to locate and analyze design changes in local subsystems).

9.5 Subversion Dependency Report

Although CHECKDEP was originally developed as an interactive tool, we found the ability to automatically generate reports about changes in the dependencies very interesting. This is implemented by adding a command-line call of CHECKDEP to the Subversion post-commit hook script, which considers the head versus the second-last revision for dependency comparison. The report contains a summary with dependency statistics (number of removed and added edges in the dependency graph).

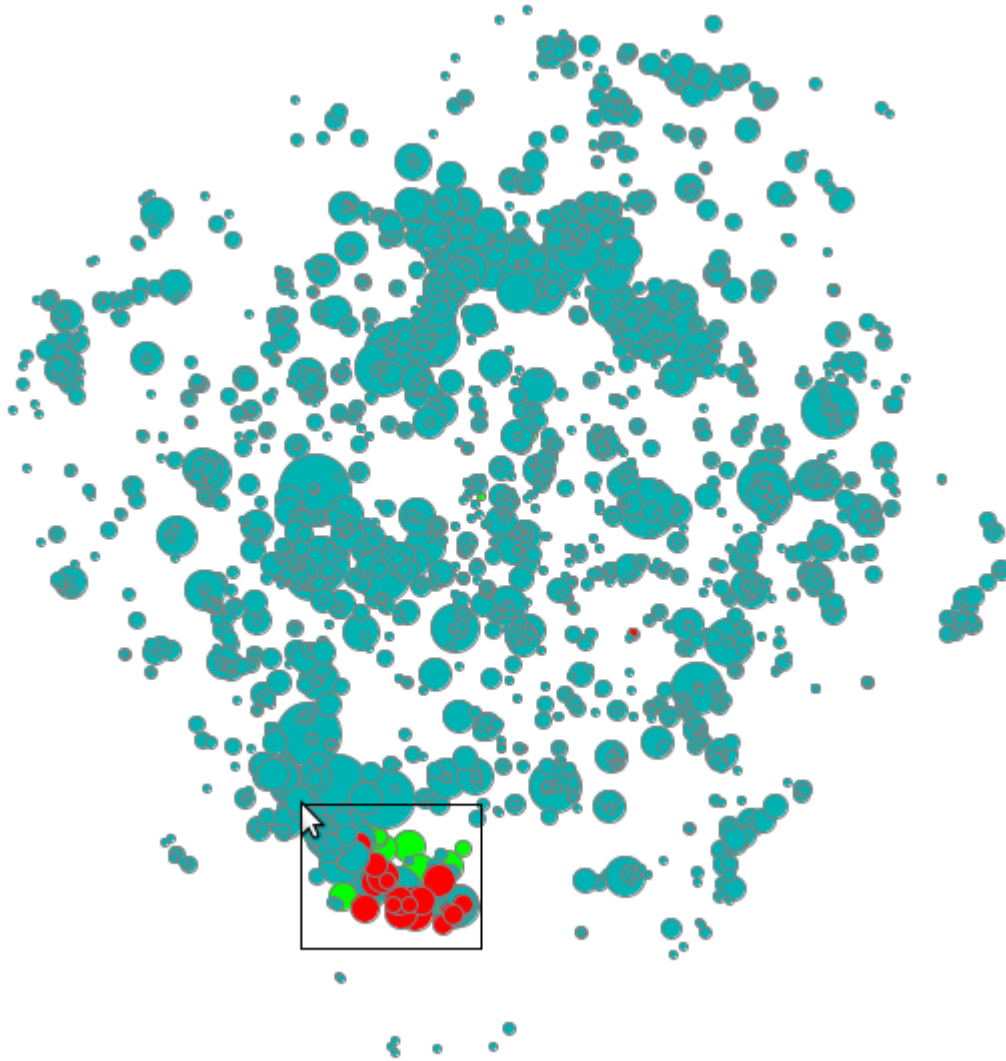


Figure 9.2: Localizing dependency changes; discs represent software artifacts (e.g. methods); the rectangle indicates a zoom area, containing most of the changed artifacts which are colored in red or green.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Dep-degree is the number of dependency edges in the use-def graph, and is defined for single program operations as well as for program functions. This indicator is easy to understand, simple to compute, flexible and scalable in its application, and independently complementing other indicators; also, it is solely based on the facts present in the program source code and can be calculated automatically.

We evaluated the proposed indicator (dep-degree) from two perspectives:

- First, we provided several examples, each consisting of two alternative implementations for the same task (e.g. a function). In each example, we showed that one implementation is easier to understand and has a better internal structure than the other one using several good reasons and strong arguments. We calculated the dep-degree values for both implementations of each example to verify whether the ‘simpler’ implementation is assigned a lower dep-degree value, meaning that it has a less complex dependency structure.
- Second, we compared the dep-degree indicator with four other widely used and well-known indicators, i.e. lines of code, cyclomatic complexity, Halstead’s difficulty and effort which are often used for measuring maintainability and understandability.

Our experiments show that dep-degree is a better indicator for readability and understandability of the code as compared with the four other indicators, and that it better reflects the improvements in the program structure.

In addition, we presented CHECKDEP, a small tool for analyzing dependency changes between different versions of software. Many software developers use a syntactical ‘diff’ in order to perform a quick review before committing changes to the repository. This simple process is lifted from the code level to the more abstract level of dependencies (i.e. dependencies between classes and functions). CHECKDEP uses a meaningful clustering layout to visualize dependency graphs which makes it a unique and notable tool among other related software.

10.2 Future Work

Dep-degree measures the number of edges in a use-def graph. Our experiments show that dep-degree is a promising and interesting indicator for code improvement and complex dependency structure. We do not claim that dep-degree is a measure for program complexity. We keep it for future work:

- to perform a careful empirical study that investigates whether the finding of our initial experiments apply in general; perhaps we can establish that dep-degree truly reflects an important aspect of program complexity and that dep-degree could be used as a complementing indicator of program complexity.
- to investigate whether dep-degree can be used to detect a need for refactoring, and to automatically infer the appropriate refactoring methods to improve the code.
- to study and extend the integration of dep-degree as an Eclipse plugin to make it available to others.

Bibliography

- [1] B. B. Agarwal, S. P. Tayal, and M. Gupta. *Software Engineering & Testing: An Introduction*. Jones & Bartlett Publishers, 2009.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] Dirk Beyer and Ashgan Fararooy. CHECKDEP: A tool for tracking software dependencies. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC 2010, Braga, June 30 - July 2)*. IEEE Computer Society Press, Los Alamitos (CA), 2010.
- [4] Dirk Beyer and Ashgan Fararooy. DEPDIGGER: A tool for detecting complex low-level dependencies. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC 2010, Braga, June 30 - July 2)*. IEEE Computer Society Press, Los Alamitos (CA), 2010.
- [5] Dirk Beyer and Ashgan Fararooy. A simple and effective measure for complex low-level dependencies. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC 2010, Braga, June 30 - July 2)*. IEEE Computer Society Press, Los Alamitos (CA), 2010.
- [6] Y. Crespo, C. Lopez, E. Manso, and R. Marticorena. Language independent metric support towards refactoring inference. In *Proc. QAOOSE*, pages 18–29, 2005.
- [7] B. Curtis, S. B. Sheppard, and P. Milliman. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proc. ICSE*, 1979.
- [8] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng.*, 5(2):96–104, 1979.
- [9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [11] G. K. Gill and C. F. Kemerer. *Cyclomatic complexity metrics revisited : An empirical study of software development and maintenance*. MIT, 1991.
- [12] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier, 1977.
- [13] M. Hammad, M. L. Collard, and J. I. Maletic. Automatically identifying changes that impact code-to-design traceability. In *ICPC*, pages 20–29, 2009.
- [14] W. J. Hansen. Measurement of program complexity by the pair: (cyclomatic number, operator count). *SIGPLAN Notices*, 13(3):29–33, 1978.
- [15] S. M. Henry and D. G. Kafura. Software-structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.
- [16] S. M. Henry, D. G. Kafura, and K. Harris. On the relationships among three software metrics. In *Proc. Measurement and Evaluation of Softw. Quality*, pages 81–88. ACM, 1981.
- [17] S. S. Iyengar, N. Parameswaran, and J. Fuller. A measure of logical complexity of programs. *Computer Languages*, 7(3-4):147–160, 1982.
- [18] C. Jones. Software metrics: Good, bad and missing. *Computer*, 27(9):98–100, 1994.
- [19] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Trans. Softw. Eng.*, 13(3):335–343, 1987.
- [20] S. R. Kirk and S. Jenkins. Information theory-based software metrics and obfuscation. *J. Systems and Software*, 72(2):179–186, 2004.
- [21] D. Kozlov, J. Koskinen, J. Markkula, and M. Sakkinen. Evaluating the impact of adaptive maintenance process on open source software quality. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 186–195. IEEE Computer Society, 2007.
- [22] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [23] M. M. Lehman and L. A. Belady. *Program evolution: Processes of software change*. Academic Professional, 1985.
- [24] W. Li. Another metric suite for object-oriented programming. *J. Systems and Software*, 44(2):155–162, 1998.
- [25] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.
- [26] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

- [27] E. E. Mills. *Software Metrics*. Curriculum Module SEI-CM-12-1.1, CMU-SEI, 1988.
- [28] G. J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices*, 12(10):61–64, 1977.
- [29] A. Oram and G. Wilson. *Beautiful Code*. O’Reilly, 2007.
- [30] S. Purao and V. K. Vaishnavi. Product metrics for object-oriented systems. *ACM Computing Surveys*, 35(2):191–221, 2003.
- [31] R. E. Al Qutaish and A. Abran. An analysis of the designs and the definitions of the halstead’s metrics. In *International Workshop on Software Measurement*, pages 337–352, 2005.
- [32] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.
- [33] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. CSMR*, pages 30–38, 2001.
- [34] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 8 edition, 2006.
- [35] F. Stetter. A measure of program complexity. *Computer Languages*, 9(3-4):203–208, 1984.
- [36] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.
- [37] K.-C. Tai. A program-complexity metric based on data-flow information in control graphs. In *Proc. ICSE*, pages 239–248. IEEE, 1984.
- [38] V. K. Vaishnavi, S. Purao, and J. Liegle. Object-oriented product metrics: A generic framework. *Information Sciences*, 177(2):587–606, 2007.