Sami Mäkelä

Cohesion Metrics for Improving Software Quality

Turku Centre *for* Computer Science

TUCS Dissertations

No ????, June 15, 2016

Cohesion Metrics for Improving Software Quality

Sami Mäkelä

To be presented, with the permission of the Faculty of Mathematics and Natural Science of the University of Turku, for public criticism in Auditorium Pub3 on May 24, 2016, at 12 noon.

University of Turku
Department of Information Technology
Informaatioteknologian laitos, 20014 Turun yliopisto

2016

Supervisors

Ville Leppänen
Department of Information Technology
University of Turku
Informaatioteknologian laitos, 20014 Turun yliopisto

Olli Nevalainen
Department of Information Technology
University of Turku
Informaatioteknologian laitos, 20014 Turun yliopisto

Timo Knuutila
Technology Research Center
University of Turku
Technology Research Center, 20014 Turun yliopisto

Reviewers

Merik Meriste
Department of Computer Control
Tallinn University of Technology
Ehitajate tee 5, 12616 Tallinn, Estonia
Estonia

Ivan Porres
Department of Department of Information Technologies
Åbo Akademi University
Domkyrkotorget 3, 20500 Åbo

Opponent

Jyrki Nummenmaa
School of Information Sciences
University of Tampere
School of Information Sciences, FI-33014 University of Tampere

ISBN 978-952-12-3388-3
ISSN 1239-1883

Abstract

Software product metrics aim at measuring the quality of software. Modularity is an essential factor in software quality. In this work, metrics related to modularity and especially cohesion of the modules, are considered. The existing metrics are evaluated, and several new alternatives are proposed.

The idea of cohesion of modules is that a module or a class should consist of related parts. The closely related principle of coupling says that the relationships between modules should be minimized.

First, internal cohesion metrics are considered. The relations that are internal to classes are shown to be useless for quality measurement. Second, we consider external relationships for cohesion. A detailed analysis using design patterns and refactorings confirms that external cohesion is a better quality indicator than internal. Third, motivated by the successes (and problems) of external cohesion metrics, another kind of metric is proposed that represents the quality of modularity of software. This metric can be applied to refactorings related to classes, resulting in a refactoring suggestion system.

To describe the metrics formally, a notation for programs is developed. Because of the recursive nature of programming languages, the properties of programs are most compactly represented using grammars and formal languages. Also the tools that were used for metrics calculation are described.

Tiivistelmä

Ohjelmistotuotemetriikat pyrkivät mittaamaan ohjelmistojen laatua. Modulaarisuus on yksi ohjelmistojen laadun olennaisista tekijöistä. Tässä työssä tutkitaan metriikoita, jotka liittyvät modulaarisuuteen ja erityisesti moduulien koheesioon. Olemassa olevia metriikoita arvioidaan ja esitellään useita uusia metriikoita.

Moduulin kohesiivisuus tarkoittaa, että moduuli tai luokka koostuu osista, jotka liittyvät toinen toisiinsa. Vastaavasti kytkentämetriikat mittaavat eri moduulien välisiä yhteyksiä.

Aluksi työssä tarkastellaan sisäisen koheesion mittaamista. Havaitaan, että luokan sisäisten suhteiden tutkiminen ei ole riittävä laadun mittaamiseen. Seuraavaksi tarkastellaan koheesiota luokan ulkopuolelta tulevien suhteiden kannalta. Yksityiskohtainen analyysi, jossa otetaan huomioon suunnittelumallit ja refaktorointi, näyttää että ulkoinen koheesio on sisäistä parempi laatumittarina. Lopuksi työssä käsitellään refaktorointimetriikoita, jotka ratkaisevat ulkoisten koheesiometriikoiden käyttöön liittyviä ongelmia. Refaktorointimetriikoiden avulla saadaan helposti selville, miten ohjelman laatua voidaan parantaa.

Metriikoiden formaalia kuvausta varten kehitetään notaatioita ohjelmien kuvasta varten. Koska ohjelmointikielet määritellään rekursiivisesti, ohjelmien ominaisuudet voidaan kompaktisti esittää formaalien kielten avulla. Tämän lisäksi kuvataan metriikkojen laskentaa varten toteutetut työkalut.

Acknowledgments

Thanks to my wife and daughter and friends and relatives. Also thanks to everybody who has helped this work, especially Ville Leppänen, Olli Nevalainen, Jukka Teuhola and Timo Knuutila.

Contents

Chapter 1

Introduction

1.1 Introduction

This dissertation is about using program analysis to improve the design of software. By improving the design of software, we mean improving the *internal quality properties* of the software. Improving these internal quality properties means that the software becomes easier to modify, maintain, or understand. Good internal quality properties should lead to good external quality.

Program analysis is a way to analyze the properties of programs. For example compilers use program analysis for optimization, and there has been a lot of research about using program analysis for detecting bugs. Because of the nature of quality, we are interested in *software product metrics* [78]. These metrics associate numerical values to software artifacts. Numerous software product metrics have been proposed to help in the software development process.

The design issues can be characterized by design rules or concepts such as modularity. In the present dissertation, we are concerned about modularity and more precisely, *cohesion* [28]. A module is called cohesive, if the components of the module are related to each other. Cohesion metrics attempt to measure how good the cohesion of a module is.

In the present work, we will perform an extensive study, where properties of several different cohesion metrics are investigated. It turns out that existing cohesion metrics are not well suited for software quality improvement. A new interpretation of cohesion that can be used for software quality improvement is therefore proposed. Finally, we present a way to automatically suggest refactorings that are related to the modular structure of programs.

In general, an important goal of the software quality research is to make the software development process more efficient. Specific goals in this dissertation include:

1. Find ways to improve the modular structure of programs.

2. Find useful ways to classify software artifacts; this would improve our understanding of software.

3. Study properties of software by the means of cohesion metrics.

4. In addition to acquiring information about the quality of programs, tools are developed to make the programming tasks easier.

We start this introductory chapter by a discussion about the central role of the software metrics in the analysis of the program quality. An important aspect here is that the metrics are related to the purpose of the software. After this, we can describe the semantic meaning of programs. In addition to the semantic meaning, programs have structure that is related to the software engineering properties of the programs. In order to describe these properties we need to consider refactorings, and how they are related to the programs. Finally we will consider how modules are used in the programming process.

In particular, the Introduction includes 10 subchapters. In Chapter 1.2 we recall the mathematical notations on sets, relations, languages and probability. The concept of software metrics is discussed in Chapter 1.3. In the software development process the development environment plays a central role. Development environments are discussed in Chapter 1.4. Software and software analysis are considered in Chapter 1.5. Better understanding of syntactical concepts in software is gained by considering refactorings in Chapter 1.6. The concept of modularity is introduced in Chapter 1.7. In Chapter 1.8 it is explained how the theoretical concepts relate to the practice of software engineering. Chapter 1.9 discusses the implementation concerns and Chapter 1.10 introduces the plan of the rest of the work.

1.2 Mathematical notations

To express our ideas, we first have to review some basic notations and concepts from set theory, automata theory [85] and probability [38].

1.2.1 Notations for sets and relations

A set or relation is usually written in capital letters such as A, while the set elements are small letters such as a. Letters u, w and v are used for words, x, y and z are numbers and fraktur font is used for elements of programs such as methods $\mathfrak{m}$ and fields $\mathfrak{f}$.

The notation $\{a \mid R(a)\}$ denotes the set of elements for which the predicate R holds.

An *ordered pair* of elements is notated (a, b). There are associated *projections* $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$. The cartesian product of sets A and B is

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Then, a set $C \subseteq A \times B$ is a *relation* between sets A and B. The application notation

$$C(a) = \{b \mid (a, b) \in C\}$$

is like function application, but it returns a set. If a set is known to include only one element a, the set can be used to mean just a. Similarly,

$$C^{-1} = \{(b, a) \mid (a, b) \in C\}$$

corresponds to the inverse function. The application notation is extended to sets by

$$C(a) = \bigcup_{a \in A} C(a)$$

1.2.2 Basics of formal languages

A *word* over an *alphabet* A is either an empty word ϵ or of the form aw, where $a \in A$ and w is a word. Concatenation of words u and w is written as uw. A *language* is a set of words.

There are different classes of languages. Probably the most important classes are the *regular languages* and the languages that can be recognized by using *Turing machines*.

A useful concept that is related to words is a *semigroup*. A semigroup is a set A with associative operator $* : A \to A$ and an identity element 1 such that $1 * a = a * 1 = a$. It is useful to consider the case where the elements of set A are functions and $*$ is the function composition operator. If we have words $w_1, \ldots, w_n$, the *free semigroup* generated by these words is the smallest semigroup that contains these words and has concatenation as $*$. A *morphism* from semigroup A to B is a function $f : A \to B$ such that $f(a_1 * a_2) = f(a_1) * f(a_2)$.

Regular languages

The class of regular languages can be defined to be the smallest set of languages that is closed under the following operations:

- Any finite language is regular.
- Union: if A and B are regular, their union $A \cup B$ is regular. Often this union is notated $A \mid B$.

- Concatenation: if A and B are regular, then

$$AB = \{wv \mid w \in A, v \in B\}$$

  is regular.

- Iteration: if A is regular, then

$$A^* = \bigcup_{n \in \mathbb{N}} \{a_1 \ldots w_n \mid w_1, \ldots, w_n \in A\}$$

  is regular. When n is 0, the notation $w_1 \ldots w_n$ is interpreted to mean the empty word.

Regular languages are also closed under intersection and several other useful operations, for example prefix(A) which stands for the prefixes of all words in A. Regular languages are also exactly the languages that can be recognized by using finite state machines.

As an example of regular languages, consider that a method $\mathfrak{a}$ can call itself or method $\mathfrak{b}$. Further, method $\mathfrak{b}$ can call method $\mathfrak{c}$, which can call $\mathfrak{a}$. The set of all possible call paths would then be $(\mathfrak{a}|\mathfrak{abc})^*(\mathfrak{ab}|\epsilon) = \text{prefix}((\mathfrak{a}|\mathfrak{abc})^*)$. It contains words $\mathfrak{a}$, $\mathfrak{aa}$, $\mathfrak{ab}$, $\mathfrak{aaa}$, $\mathfrak{aab}$, $\mathfrak{abc}$, $\mathfrak{aaaa}$, $\mathfrak{aaab}$, $\mathfrak{aabc}$, $\mathfrak{abca}$ etc.

Grammars

A common way to define languages is by using *grammars*. In a grammar, we have a set of *terminals* Σ and a set of *non-terminals* N. Then we have a set of *rules* that are pairs $(u, v) \in (\Sigma \cup N)^* \times (\Sigma \cup N)^*$, denoted as $u \to v$. Given a word w, if $w = w_1 u w_2$, it can be rewritten with rule $u \to v$ to $w_1 v w_2$. The language defined by the grammar is then the subset of Σ^* containing words that can be derived by applying the rules of the grammar beginning from the *initial nonterminal* $I \in N$.

For example, the above language prefix($(a|abc)^*$) would be given by the grammar

$$\begin{array}{rcl} I & \to & aI \\ & | & abcI \\ & | & ab \\ & | & \epsilon \end{array}$$

In general, grammars can represent more languages than regular languages. In fact, they can generate the languages that are recognizable by *Turing machines*.

Turing machines

Turing machines provide a mathematical foundation for computer science and software engineering. A Turing machine represents an idealized digital device. There is no limitation of resources and no failures. The two parts of a Turing machine are the tape that represents the memory, and a finite state machine that reads from and writes to the memory. A *universal Turing machine* is then an idealized computer: after programming, it can simulate the operation of any Turing machine.

This can be tied up with Section 1.3.1, where the requirements for a software system are considered. Because the requirements might change, the behaviour of the system should always be modifiable. We can for example use different tapes for different users of the system.

The nice aspect of Turing machines is that they show exactly what can be done with digital devices. The other side of this coin is that many things such as deciding whether two programs have equivalent semantics are undecidable.

1.2.3 Probability

Formally, let A be a set of *events*. Then the *probability* $P(a)$ of an event $a \in A$ fulfills the following axioms:

- $P(a) \in [0, 1]$.
- Unit measure: $P(A) = 1$ and $P(\emptyset) = 0$.
- For a countable sequence of pair-wise disjoint event sets a_1, a_2, ... we have $P(\bigcup_i a_i) = \sum_i P(a_i)$.

The intuitive interpretation of $P(a \cup b)$ is the probability that either event a, b or both happen. The interpretation of $P(a \cap b)$ is that both a and b happen. Two events a and b are *independent*, if $P(a \cap b) = P(a)P(b)$.

A very useful concept is *conditional probability*. It is defined

$$P(a \mid b) = \frac{P(a \cap b)}{P(b)}$$

The intuitive interpretation is the probability that a happens, if b has happened. Note that there is no assumption of causality between a and b.

1.3 What are software metrics?

Software metrics [39] are simply numerical values associated to different aspects of software or software development process. First, there is the dynamic view to the software, where the software is measured at run-time.

The measurements can be extended to the effects to the environment, for example how many users the software has, and how well the software fulfills its requirements. Second, we can measure how the software is changed. This includes measuring the software development process. These two kinds of metrics correspond to the external and internal quality attributes of software. Third, the properties of software artifacts can be measured. These metrics are called software product metrics. A software artifact can be a model, design document or source code for the program.

In the present study, we are interested in using software product metrics to improve software quality, more specifically the internal quality. In this improvement process one must first have a clear understanding of interactions between external and internal quality attributes, and how software product metrics and quality can be connected together.

Considering external measures, we are interested in measuring how well the software fulfills its requirements. If the internal quality is good, it should be easy to modify the software so that its external quality is improved. Software product metrics and internal quality can be connected using the concept of *programming tasks*. The tasks are use cases for the software environment. If programming tasks are easy to perform, the internal quality tends to be good. On the other hand, it is possible to show dependencies between a programming task and software product metrics. Using these tasks, one can define a cost model for a software artifact.

The research and practice of software engineering has been mostly concentrated on finding how to design the software instead of improving the productivity of the software environment. Probably a lot of what is needed on software environments is the same as on other user interfaces. One can argue that better identifying the tasks needed in software development helps finding the issues that are specific to software engineering.

1.3.1 Process view

We can describe the above ideas in the following way. The starting point is the set of possible requirements (or environments) R. For fulfilling R there is a set of possible software systems S. The requirements and the software change, when time elapses. To describe this, let $r : \mathbb{R} \to R$ and $s : \mathbb{R} \to S$ be functions for requirements and software on a given point of time. A function $f : R \times S \to \mathbb{R}$ tells how well the software fulfills its requirements (in production use). Negative values mean that software is useful, and positive values mean that software is harmful wrt. requirements. The effort of changing software depends on the previous state of the software, and on the particular change, so it can be modelled using function $e : S \times DS \to \mathbb{R}$. Let $c(t_0, t_1)$ be the total cost of the software in the time interval from t_0 to t_1. (The set DS is the set of changes to S.) If $s'(t)$ represents a change made

to software at time t, the value of c is the the sum values of $f(r(t), s(t))$ and $e(s(t), s'(t))$ when $t \in [t_0, t_1]$.

The cost can be defined to be some empirically measurable property of the software system like the money or hours spent on developing the software. As such, it is intended to be the connection point of software process metrics and software product metrics. If the cost can be split to several pieces from which it can be calculated, perhaps a better way to approximate it can be found.

The changes DS above can be thought to be the set of programming tasks. Clearly effort is needed to make progress on the programming tasks. Also the current system state can be calculated from these performed tasks.

A problem in defining the improvements of software is that the way the requirements change can be affected by the changes made to the program. The software development process can thus be divided into two interacting parts, where the change depends on the external part and the internal part.

$$\begin{aligned} s'(t) &= q(r(t), s(t)) \\ r'(t) &= p(r(t), s(t)) \end{aligned}$$

where q describes how the software changes, and p describes how the requirements change, in a given state. Now, given f, p, and e, we should find the best possible q. This means that based on the knowledge we have, we should find an optimal way to modify the software.

The two interacting parts correspond to two ways of improving the software. The first way to improve a software system is to change it to fulfill its requirements better. This is related to the semantics of the system. New requirements might necessitate the addition of new semantic content. The second way is to change the software in such a way that future changes of the software will be facilitated. This approach is related to the internal structure of the program. Because changing requirements are hard to predict, most work in this question has been concentrated on the internal structure of programs. The changes that only affect the structure of programs are called *refactorings* [43].

The existence of refactorings implies that programs include content with no semantic meaning, which we call the internal structure of the program. In a wide sense, this content can be seen to include code documentation, models and other domain knowledge. To determine what kind of changes should be made to the internal structure, it is not enough to consider, how good the internal structure is at a given moment of time. Also the process aspect needs to be taken into account. For example, because changing the model necessitates extra design and programming work, unimportant changes should not be made. If there are several programmers involved, radical changes of the model can be unwanted, because that requires relearning the program structure.

1.3.2 Improvement of the internal quality

Refactoring can be used to improve the design of programs. For this assume that it is possible to isolate a piece of internal structure of a software system into a model. For each model, there is a set of possible models that can be reached by refactoring the program. The problem of finding a pertinent way of refactoring the program is reduced to finding a model that is better than the current one.

A simple way to find good refactorings would be to define the cost of models, and then attempt to find a better model than the current one. The problem here is that it might be hard to define the cost of a particular model. Often it is attempted to improve the design of programs by using *design heuristics*. These heuristics stand for rules for making development decisions. These rules are found by experience gained from software development. Two important examples of this kind of rules are *code smells* from refactorings [43] and *forces* from design patterns [44].

The ideas of design heuristics can be formalized by using *software metrics*. The intention is that a software artifact gets low metric values if it agrees with a design heuristic, and it gets high values in the opposite case. For example, the metric LCOM [26] tries to describe the design rule for cohesion. Now, as there are several design heuristics, and many software metrics, the problem is how to combine them.

To describe how these metrics can be combined, we give the following framework. Assume there is a metric $\mu : A \to \mathbb{R}$ that is intended to represent some design heuristic. Then there is an associated partial order $\sqsubseteq_\mu$: $A \times A$ between software artifacts

$$a \sqsubseteq_\mu b \Leftrightarrow \mu(a) \leq \mu(b)$$

For example, $a \sqsubseteq_\mu b$ means that according to the heuristic represented by the metric μ, model a is better than b. With partial orders, the relations between different metrics can be defined. The partial order $\sqsubseteq_\nu$ *agrees* with a set of partial orders $\sqsubseteq_{\mu_i}$, if

$$(\forall i : a \sqsubseteq_{\mu_i} b) \Rightarrow a \sqsubseteq_\nu b$$

If the partial order of ν agrees with partial orders μ_i for all i, then ν is a *valid combination* of design heuristics μ_i.

In the simple case, where metric μ is defined in terms of metrics ν_i, that is, it is a function $\mu(\nu_1, \ldots, \nu_n)$, it is a valid combination of the metrics if

$$\frac{\partial \mu}{\partial \nu_i} \leq 0$$

In the ideal case, a design heuristic should agree with the cost of the model. In practice, to determine whether a software metric implements a useful design heuristic, at least the following aspects need to be considered.

- When the metric gives an alarming value, there should be a design problem.
- The design problem can be found easily based on the metric values. For example if metric values always indicate a similar design problem, it can be found easier.
- The design problems that are found using the metric are not found easier by some other metrics.

If two metrics give similar results, the one that is simpler and easier to understand should be used.

1.3.3 Prediction

Software metrics are often used for prediction in software processes [14]. Assume that there is a process metric μ that would be useful to estimate. Further assume that there is a set of metrics ν_i that can be easily calculated. Now, linear regression or some other statistical method can be used to get a formula approximating μ from ν_i.

The only requirement for this kind of metrics is that they are independent of other metrics. (Otherwise they would be redundant.) This can be confirmed using statistical analysis. The most important and obvious metric for prediction is size of the program. Correspondingly, for design heuristics, we could have a metric for sizes of modules.

Even if correlation is found to exist between a metric and an external quality attribute, it cannot be deduced that this metric could be used as a guideline for software quality improvement. For example, it is generally thought that as the software matures, the internal quality and maintainability of the software become lower while external quality attributes become better. When this kind of studies have been made with different software packages, it has been found out that different metrics have had the best correlation with software quality.

1.3.4 Anomaly detection

Above we presented the top-down approach to software metrics, where design heuristics are used as a basis for software metrics. An alternative, bottom-up approach to software metrics is developed in this section. In this approach, one starts from the basic properties of software artifacts, and tries to derive design rules for good software from them.

Code smell [43] can be defined to be a binary predicate that is applied to some part of the program. For each predicate R, the probability that a random software artifact has that attribute is $P(R)$. In addition to predicates it can be convenient to have a random variable V in some finite set that

tells some property of a part of the program. Then the probability that for a software artifact the property V has value a is $P(V = a)$. For numerical variable X is a numerical variable, notations $P(X = n)$ and $P(X < n)$ are used. More generally, it is possible to define a distance d that can be used as $P(d(y, X) < t)$ to find out what is the probability of X being close to the point y.

The idea of finding anomalies in software is to build a model for predicting the properties of a program block. Assume that according to this kind of model, the probability of X being near y is $P(d(y, X) < t) = p$. The experiment X is repeated n times, i.e. the metrics set X is measured for n different programs. Assume that m of the total n program parts were close to y. According to our model, the probability that m items are near y is $p^m(1-p)^{n-m}$ and so the probability for less or equal to m items

$$p^* = \sum_{i=0}^{m} p^i(1-p)^{n-i} = (1-p)^n \sum_{i=0}^{m} \frac{p^i}{(1-p)^i}$$

If probability p^* is small, it is an indication of the presence of an anomaly. Based on the model, there should be more programs with metrics values that are near y. The system has a design heuristic that causes programmers to avoid this kind of programs.

The approach presented here is similar to approaches using data mining to find programming patterns. Errors can be detected based on these patterns [53]. The detectors for these patterns can be thought to be very specific metrics.

Both top-down and bottom-up approaches have problems. One of the problems is that the programs have semantic content that is not related to the quality of programs. To solve this problem, one can consider models that include knowledge about the software, and estimate the usefulness of this knowledge. These models can be found by using refactorings. It is easier to estimate the usefulness of smaller models than that of the whole program.

1.4 Software Development Environments

As seen above, based on the requirements of software, the software can be divided into two parts: the actual software (called the *production environment*), and the *development environment* that is used to modify the software. The development environment is about programs, and it is known precisely what programs are, therefore justifying and defining the field of software engineering.

The program analysis and metric tools described in this dissertation belong to the development environment, so it is important to understand what

the development environment is, and what is the purpose of the development environment. Another reason why development environments need to be handled is that programming tasks are performed using the development environment. If the development environment has more automation, the costs of programming should be lower.

Given a development environment, any kind of software can be built. On the other hand, some kind of programs are easier to write in some language A than in another language B. This issue becomes rather confusing, when one notes that it is possible to define the syntax and semantics of the language A in language B. This means that in theory, it does not matter what kind of language is used. Or perhaps it means that an environment which supports multiple languages is to be preferred. One conclusion that can also be drawn is that the size of a program depends on the semantics of the program (Kolmogorov complexity [57]).

1.4.1 Development environment and programming language

The development environment has to support the following two tasks. First, it is used to create the production environment, or at least the executable, which is the basis of the production environment. Second, it is used to modify the production environment, or create new versions of the executable. Next we review the history of development environments, and possible future directions.

Development environments are traditionally built around a programming language. The *source code* is converted into an executable by using a *compiler*. The source code is a collection of text files, which are edited using a text editor. Searching information in source code is done by standard text search tools. Nowadays there are also tools that use structural representation of the programs, or type information for searching information.

A debugger is a tool that lets the programmer inspect a production environment carefully. Another requirement for the debugger is that it links the concepts in the production environment with the concepts in the development environment.

These tools are collected to form an *integrated development environment* which may additionally include design tools and project management tools.

Because the source code is in a textual format, it is hard to make modifications to it. For this reason different kinds of refactoring tools have been developed.

Over time, both development environments and programming languages have grown more and more complex. Often the development environments seem arcane and are more cumbersome to use than the systems that are developed using them.

1.4.2 Capabilities of development environments

The capabilities of development environments are related to software quality. To understand this relation, one has to consider some important programming tasks, and their relation to capabilities of programming environments.

Refactorings are often used to improve the design of programs. Development environments include support for performing refactorings. These operations are conceptually simple, but because programs are stored as texts, they become complex. For example Eclipse [32] includes support for renaming variables, methods, and classes, moving methods, etc. to aid the refactoring operations.

Another important feature is searching for needed methods or classes. This is usually done using type information, or information about modules. For classes, they can be searched by name. Once the class has been found, the method can be selected from a list of names of methods of the class. The more the class has methods, the harder it is to find the correct method.

The capabilities of development environments have an impact on software design. For example, because current development environments have automatic refactoring tools, one should not anymore be focused on design rules such as use getters/setters instead of public fields, or use virtual methods instead of typecase. These can simply be considered as different alternatives that can easily be switched into another one by applying refactoring.

1.4.3 Knowledge and software engineering

As an extended viewpoint, software can be understood as *knowledge* about tasks that the computer is supposed to perform. In addition, the software developer needs knowledge about the software itself. This knowledge belongs into the development environment.

To build or understand software, many types of knowledge are needed. At least the following ways to encode knowledge about software exist currently:

- Source code.
- Comments.
- Types.
- Design documents.
- Models (for example UML [90]).

Most of the knowledge is encoded using the programming language, which is by far the most important tool in software engineering. Models

often have knowledge that overlaps with the programming language. Comments are informal, and because of this, they might be incorrect, and cannot be used as input to automated tools. Types have sometimes been said to be formal kind of comments.

It would be convenient to have knowledge about design decisions included in the source code. For example, the design decisions are often made for efficiency reasons. If this is not documented, one may suspect that a design error occurs. Because both simple and efficient solution have their advantages, in the ideal case both of these should exist. Knowledge about *design patterns* [44] can also be useful. Patterns are elements of the programming style that is not directly supported by the programming language.

1.5 What is software?

The run-time behaviour of the software, and the connection between the behaviour of software and software artifacts are shortly discussed in this section. The behaviour of a program artifact is often called the *semantic meaning* of the artifact.

Semantics of the programs are considered here for two reasons. Obviously the programming tasks are related to the semantics of the programs. If a programming task is supposed to improve the external quality of programs, the semantics of a program will be changed. Also the concept of refactorings is related to the semantics of programs.

The behaviour of software can be observed on several levels. The most important level is observing the impact of the software on users and the organization. This impact can be compared to the requirements of the program.

A naive way to define the semantics of programs would be by stating the input and output of the program. In this view it is assumed that the program acts as a function. This function can be partial or non-deterministic. It is usually assumed that the relation between input and output can be described using Turing machines.

Notice that changing the input-output semantics does not necessarily change the impact of the program, because the changes could have been made for an input that never occurs. On the other hand, changes that only change the performance, but not the input-output semantics, can have large impact if the program has real time requirements.

When looking at different parts of the program, one can observe the state changes in the program. It is assumed that a computer is running the program, and the changes of the content of the memory of the computer are recorded. The operational semantics of programming languages usually works on this level to define the input-output semantics of a program.

One of the most important roles for semantics of programs in studying programming tasks is finding the relationships between the different parts of the program. For example, if a part is modified, the semantics of the program can be used to find out which parts of the program need to be tested. It is easy to see that semantic relationships between software artifacts can have a drastic effect on the efforts needed for performing programming tasks. Finding this kind of relationships is a part of the field of program analysis.

It is interesting to consider how one should express the possible observations that can be made about the state transitions. These observations are related to concrete parts of the programs, so one has to relate the observations and the parts of the programs. For this purpose, it is enough to think the program as an array of characters. The program parts are then subsets of these array indexes.

1.5.1 Semantics of programs

A program in itself is a formal entity and has well defined semantics as expressed by Turing machines or some similar formalism. This degree for formality is a consequence of programs being interpreted by machines.

There are many ways to describe the semantics of programs in mathematical terms. The problem in programming language semantics is that it is hard to state the meaning of programming language constructs. To illustrate, what makes defining the mathematical semantics of programs difficult, it is useful to consider the following example. It would be natural to model the functions of a program using set-theoretical functions. But if the language supports higher-order functions, a function can take itself as an argument. The set-theoretical model for this kind of functions would then be a set that contains itself.

A popular approach for formal definition of programming languages is operational semantics [76], where each programming language construct is associated with a state transition. For purposes of the present study, it is useful to consider the set of all possible executions or *runs* of the program. This set can be thought to define the semantics of a program. A run can be expressed as a sequence of states or state transformations of the program. Moreover, each run can be given a probability. This model of semantics corresponds to observing what the computer does at the run time. The state transformations can be divided into two parts: internal and external. Internal transformations are performed by the computer, and the external ones are performed by the environment. For example user input belongs to the external transformations. From the state transformation, it is possible to deduce input and output of the program.

The programming languages can be thought to have two kinds of non-determinism: first, the non-determinism caused by the environment, and

second, the internal non-determinism that is caused for example by concurrency. If all non-determinism is thought to be external, then the internal part can be simulated by a Turing machine.

To formally represent the possible state changes, a set of *atomic state transformations* can be used. Atomic transformations are either external or internal. Then the set of all possible state transformations is the free semigroup of these transformations. Obviously the free semigroup of atomic internal transformations should describe transformations that can be defined using Turing machines. It can be required that each transformation corresponds to a unique sequence of atomic transformations.

1.5.2 Probabilities and relationships

In theory, the runs of a program can be derived from the definition of the programming language. To define the environment, one has to deal with probabilities of external events.

Empirically, each run of a program is associated with a probability. Moreover, for our eventual purpose, it is more important to consider how the programmer inspects the program than how the computer executes it.

1.5.3 Flow of control and data

For program analysis, exact semantics of the programs is not always needed. That level of precision is reserved to the field of proving program correctness. Atomic state transformations can be divided into equivalence classes. For example, instead of whole program state, one might only be interested in which method the control is at a given state, that is the *control flow* [4]. Another common possibility is to analyze the *data flow* [87]. Data flow is about how the information flows between variables and expressions in the program.

The concepts from automata theory are useful in program analysis [36, 65]. Adapting these ideas, we use the following general approach to program analysis: the type of the analysis is represented by alphabet Σ which represents the possible states or locations of the program. Then there is a language $L \subseteq \Sigma^*$ that represents the program being analyzed. Each word in the language represents a possible sequence of states in the program. Then another language $P \subseteq \Sigma^*$ represents a property of a program. The intersection $L \cap P$ is calculated to see if the program has the property.

To speak about control flow, it is necessary to fix the locations where the control can go through. For this, there is a set of control locations $\mathfrak{A}$. Then the set of possible runs B is a subset of all sequences of locations $\mathfrak{A}^*$. If the locations are procedures, there are two ways of moving the control: either a procedure is called, or a procedure exits. We can say that $\mathfrak{A} = \{\mathfrak{m}^{\downarrow}, \mathfrak{m}^{\uparrow} \mid$

$\mathfrak{m} \in \mathfrak{M}\}$, where $\mathfrak{M}$ is the set of all procedures, $\mathfrak{m}^{\downarrow}$ means calling a procedure $\mathfrak{m}$, and $\mathfrak{m}^{\uparrow}$ means returning to method $\mathfrak{m}$. Let us assume that the control always returns to the caller and that the first called procedure is $\mathfrak{m}'$. This can be described by a grammar

$$\begin{aligned} B &\rightarrow \mathfrak{m}'^{\downarrow} B_{\mathfrak{m}'} \\ B_{\mathfrak{m}} &\rightarrow (\mathfrak{n}^{\downarrow} B_{\mathfrak{n}})^* \mathfrak{m}^{\uparrow} \end{aligned}$$

Now it is possible to define the set of all call stacks for a control sequence $w \in B$. First we define morphism stack from control transformations to stack state transformations

$$\begin{aligned} \text{stack}(\mathfrak{m}^{\uparrow})(u\mathfrak{m}') &= u \\ \text{stack}(\mathfrak{m}^{\uparrow})(u) &= u\mathfrak{m} \end{aligned}$$

The first transformation pops the last element $\mathfrak{m}'$ from the stack $u\mathfrak{m}'$, and the next transformation pushes the called method $\mathfrak{m}$ to stack $u \in \mathfrak{M}^*$. The set of all call stacks is then

$$\text{stack}(\text{prefix}(w))(\epsilon)$$

where prefix(w) is the set of *prefixes* of word w, that is, $w_1 \in \text{prefix}(w)$ iff there exists word w_2 such that $w = w_1 w_2$.

The language containing all possible call stacks helps us to determine the control relationships between the methods. In practice, some simplifications are still needed. Let *call graph* $\mathcal{C}$ be a directed graph that has methods as its vertices. There is an edge from method $\mathfrak{m}_1$ to $\mathfrak{m}_2$, if it is possible that method $\mathfrak{m}_1$ calls method $\mathfrak{m}_2$. We can define the language $B_{\mathcal{C}} \subseteq B$ containing all possible call stacks by grammar

$$\begin{aligned} B_{\mathcal{C}} &\rightarrow M_1 \mid \ldots \mid M_n \\ M_i &\rightarrow \mathfrak{m}_i\{M_j \mid (\mathfrak{m}_i, \mathfrak{m}_j) \in C\} \end{aligned}$$

Note that this language can also contain runs that are not actually possible.

Basic data flow relationships are more complex than basic control flow, because there are expressions, fields and variables that all need to be taken into account. Therefore expressions, fields and variables are used as locations, and the data flow between them defines the possible state transformations.

Data flow and control flow can have complex relationships. For example, different stacks might have different data flow possibilities.

1.5.4 Interfaces and control flow

As an example of the relation between data flow and control flow, consider methods, that are called from an object of a known class. The program

locations Σ are now class-method pairs. Then the internal calls for class $\mathfrak{c}$ are

$$I_{\mathfrak{c}} = (\mathfrak{c}, \mathfrak{M})^*$$

The call graph can now be constructed similarly to the more coarse-grained algorithm, but this-calls can be resolved more precisely. A way to further classify objects is using creation sites. Then points-to analysis could be used for finding out the possible transitions.

There are several ways to define how a method $\mathfrak{m}$ in a class $\mathfrak{c}$ might access a field $\mathfrak{f}$ in the same class $\mathfrak{c}$:

$$\begin{aligned} A_1(\mathfrak{c}, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}, \mathfrak{m})(\mathfrak{c}, \mathfrak{f}) \\ A_2(\mathfrak{c}, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}, \mathfrak{m}) I_{\mathfrak{c}}(\mathfrak{c}, \mathfrak{f}) \\ A_3(\mathfrak{c}, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}, \mathfrak{m}) \Sigma^*(\mathfrak{c}, \mathfrak{f}) \end{aligned}$$

The definition A_1 is direct access, A_2 is internal access, and A_3 is any access. Similarly, for an external field access, where a method $\mathfrak{m}$ in a class $\mathfrak{c}_1$ might access a field $\mathfrak{f}$ in another class $\mathfrak{c}_2$, we have

$$\begin{aligned} B_1(\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}_1, \mathfrak{m})(\mathfrak{c}_2, \mathfrak{f}) \\ B_2(\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}_1, \mathfrak{m}) I_{\mathfrak{c}_1} I_{\mathfrak{c}_2}(\mathfrak{c}_2, \mathfrak{f}) \\ B_3(\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}_1, \mathfrak{m})(I_{\mathfrak{c}_1} I_{\mathfrak{c}_2})^*(\mathfrak{c}_2, \mathfrak{f}) \\ B_4(\mathfrak{c}_1, \mathfrak{c}_2, \mathfrak{m}, \mathfrak{f}) &= (\mathfrak{c}_1, \mathfrak{m}) \Sigma^*(\mathfrak{c}_2, \mathfrak{f}) \end{aligned}$$

where B_1 is direct call, B_2 is the case where $\mathfrak{c}_1$ calls $\mathfrak{c}_2$ internally and $\mathfrak{c}_2$ calls the field internally, B_3 is the case where all call chains that only involve $\mathfrak{c}_1$ and $\mathfrak{c}_2$ are considered, and B_4 is any call. A problem in defining software product metrics is determining which of the possible definitions is the best.

1.6 Refactoring

Changes aiming at improvements in the internal quality of software should not change the semantic properties of the software. These kind of changes are called *refactorings* [43]. Refactoring does not affect the external quality. We say that the refactorings change the *structure* of the software. It is obvious that when talking about improving internal quality, the question is about refactorings and the structure of the software.

Refactorings and changes that affect the semantics of programs are not completely disjoint. It can be thought that changing the semantics of a program is divided into two parts: first the program is refactored so that the change is easy to perform, and then the change is performed. Also design of a program is not completely independent of semantics: different designs contain different kinds of semantic possibilities. In fact it is impossible to completely separate the structure of the program from its semantics. Otherwise there would be only one way to write a program with given semantics.

In this section, we propose a formal approach to refactorings. In this approach, refactorings are linked to *structural models*, which include non-semantic knowledge about programs. The purpose of these models is then evaluated. In addition, the goals of the described refactorings are discussed.

The structural models are related to the internal models of programs in the development tools. For example a compiler might have for a program several different internal representations, each of which being closer to the hardware representation of programs. It can be thought that the hardware representation has a minimal amount of structure that would be helpful to programmers, and has almost purely semantic content.

The internal representations of programs in compilers are examples of data structures that are used to represent programs in development environments. Other examples include

- Type checkers.
- Program query tools. It is easy to convert programs into general semi-structural data that can then be queried using general tools.
- Program editing. Most program editing is currently done using the textual representation. For example refactorings would be easier to implement using other kinds of data structures. In model driven development, graphical representations are used instead.

In this section we are interested in finding data structures, for which it is easy to perform refactorings and other modifications to the software. It will be seen that this kind of data structures can be considered as dual to the internal representations used in compilers. These data structures contain only information that is used to help programmers, and have no semantic content.

1.6.1 Formal definition of refactoring and related concepts

Several approaches to formally define refactorings exist[66, 55]. In this section we adapt these approaches to focus on the non-semantic structure of programs.

Consider a set of programs A. Denote the semantic equality of two programs a and b by

$$a \equiv b$$

Then a refactoring relation is denoted by $a \mapsto b$, where $a \equiv b$. This means that a can be refactored into b. It is required that the refactoring relation is reflexive, transitive and symmetric. These requirements imply that the refactorings have associated equivalence classes $[a]$. The set $[a]$ is just the set of programs that can be reached from a by applying refactorings.

Given a refactoring, a programming language A can be divided into two parts B and C. The language B contains the structure related to the refactoring, and from the language C this kind of structure has been removed. If we have a program $a \in A$, there is a corresponding model $b \in B$. A refactoring is now any change of the model b. We need to show how the changes in model are connected to changes in the original program.

More formally, there is a bijection $f : A \to B \times C$ such that if $f(a) = (b, c)$, then $a \equiv c$ and for each $[a]$, $\pi_1(f([a])) = c$. (Here π_i is the projection $\pi_i((a_1, ..., a_n)) = a_i$.) In other words, the function f converts a program a to two parts b and c, where c is semantically equivalent to a. If a refactoring is applied to a, the program c will stay the same. The program b has to change because f is bijective. If the component b is changed, the new a is uniquely determined by f. The problem is that if c is changed, how should b and a change? First note that a uniquely determines b. Therefore only changes induced to a have to be considered.

Now the idea is that the function f gives an editable view b of a. In general, the function f does not have to be a bijection. It needs to fulfill the rules for *lenses* [42]. The theory of updateable views has been used as a starting point for some research in model synchronization.

In the current case, an *updateable view* of programs $g : A \rightharpoonup B$ consists of two functions $g_\to : A \to B$ and $g_\leftarrow : R_g \to A$. Here the set $R_g \subseteq A \times B$ determines the valid modifications. The following rule holds for R_g: If $g_\to(a) = b$, then $(a, b) \in R_g$. It is said that if $g_\to(a) = b$, then b is a view to program a. Assume that view b is changed to b' and $g_\leftarrow(a, b') = a'$. Then a was updated into a' using the updateable view. Following lenses [42], we get the following rules for updates:

$$\begin{aligned} g_\leftarrow(a, g_\to(a)) &= a \\ g_\to(g_\leftarrow(a, b)) &= b \\ g_\leftarrow(g_\leftarrow(a, b'), b) &= g_\leftarrow(a, b) \end{aligned}$$

Dividing language A can now be formulated as two updateable views $g : A \rightharpoonup B$ and $h : A \rightharpoonup C$, for which $h_\to([a]) = c$. All modifications are valid for h: $R_h = A \times C$. For g, the modifications cannot change the equivalence class:

1. If $a \mapsto a'$, then $R_g(a) = R_g(a')$
2. If $g_\leftarrow(a, b) = a'$, then $a \mapsto a'$.

This restriction to R_h ensures that updates to B represent valid refactorings. Now we can write the bijection f as

$$f(a) = (g_\to(a), h_\to(a))$$

Given c and b, we can uniquely determine a. This implies that

$$g_{\leftarrow}(a, b) = f^{-1}(b, h_{\rightarrow}(a))$$

Now we can remove A from the equations, and see what is the relationship between B and C. If we now only have b and c, then changing b cannot change c. Given b and c, there is a set $B_{b,c} = g_{\rightarrow}([f^{-1}(b, c)])$ corresponding to possible refactorings, and b can be changed inside this set. So the first form of change is

$$(b, c) \mapsto (b', c)$$

where $b, b' \in B_{b,c}$. On the other hand, if c is changed to c', we get $a = f^{-1}(b, c)$ and $a' = h_{\leftarrow}(a, c')$, and finally $b' = g_{\rightarrow}(a')$. Therefore the second form of change is

$$(b, c) \mapsto (g_{\rightarrow}(h_{\leftarrow}(f^{-1}(b, c))), c').$$

For this kind of system, we need two components: the relation R, which tells how b can be changed, and function $h(b, c, c') = g_{\rightarrow}(h_{\leftarrow}(f^{-1}(b, c)))$, which tells how structure b is changed when semantics c is changed.

There should be a metric that tells how much b is changed. The functions should be selected so that b changes as little as possible.

Once the program has been divided into models, one has to consider for each of these models, how it is used to help in program development. These *usage patterns* can then be used to define the effect of the model in the software development process.

1.6.2 A simple language

To show how to use the above framework, define a simple language $L \subseteq \Sigma^*$. First the lexical structure of the program is ignored by associating with each $s \in \Sigma^*$ a sequence of symbols S^*. These symbols are either builtin symbols (set B) or names (N); $S = B \cup N$.

The syntax of L can be defined using a grammar:

Program	$\rightarrow$	Module*
Module	$\rightarrow$	module N { Method* }
Method	$\rightarrow$	method N [N^*] { Command* }
Command	$\rightarrow$	Expression ;
	$\mid$	{ Command* } ;
	$\mid$	N = Expression ;
Expression	$\rightarrow$	N
	$\mid$	Expression [(Expression ,)*]
	$\mid$	lambda [N^*] { Command* }

Features such as reference cells and control structures are thought to be defined as builtin functions. For this, there is a set of builtin function names $F \subset N$.

We consider the following kinds of structures in the program:

- Lexical level, indentation, style issues.
- Names and scoping.
- Modules. The order of the items inside modules can be changed. The items can be moved into different modules.
- Lambda expressions (anonymous functions) can be moved into top-level.
- Sequences for commands: flatten them.
- Expressions can be flattened using temporary variables.
- The level of generality for functions: remove extra generality.
- Also remove functions with only one parameter.

With more semantic knowledge about builtin functions, new refactorings can be defined.

1.6.3 Lexical level

The lexical level of a programming language is the most established example of what we are trying to achieve. By ignoring details such as indentation and other whitespaces, it is possible to concentrate on syntactical structures. The indentation is very important for understanding the source code.

The problem is determining how the lexical structure should be changed when the program has been modified on syntactic level. The previous style of program should be preserved as much as possible. In the worst case, if something is changed, the old indentation is completely ignored. An easy way would be to automate indentation. Manually made changes would be removed. One can define a distance for programs that measures, how close the indentation structure is to the original.

In current development environments, the indentation has been mostly automated. New parts of the programs can be given indentation and white-spaces by using a *pretty-printer*, that is applied to the syntactical structure of programs.

Perhaps the easiest way to implement the internal representation is to make the lexical tokens to be "objects". If a new object is added, it will have new indentation. One could now check that the values of the old objects have not been changed. So there are lexical objects, O, and these objects are in a linear order. The lexical objects have associated values and associated indentation properties.

The generation of this structure from the textual representation of the program is the task of a *lexer*. The next step is *parsing*, where lexical tokens are arranged in a tree structure. This tree structure is completely determined by the grammar of the language, and each structure can only have one possible sequence of lexical tokens, so this phase does not need any additional considerations.

1.6.4 Names

The next important style issue is the naming of variables and other software artifacts. This issue does not have as clear-cut solutions as indentation issues, because there cannot be a universal naming scheme for software artifacts.

There are two things which tend to do the situation more complex. The first one is that if the programmer wants to change a name, it has to be changed everywhere, because there is not an object in the textual representation that would represent the software artifact. Another complication is that the programmer might want to give several different names or *aliases* to a software artifact. This overlaps with the semantic concept of pointers or references. Probably the closest thing to this is that name declarations can be accompanied by comments that describe the purpose of the artifact.

It is not a great problem that local variables cannot have several names. Global named artifacts such as fields and methods can quite easily have several different names. First, fields can have getters and setters so that this is reduced to the case of methods. A method can have another method with a different name that simply calls the original method. Unfortunately this workaround might have an overhead on the run time. For classes, the situation is not as easy.

The lexical objects that represent the same name are replaced with a name object. In this model, the name objects can be given new values. The scoping rules generate restrictions for these changes. This seems easier than indentation: If a new name object is added, it needs to be given a new name value. For example Eclipse has objects that correspond to named software artifacts, but are these used for refactorings?

The naming related issues are probably the clearest example of problems connected to the use of text based program editing.

1.6.5 Expressions and temporary variables

In general, there are three ways to use local variables.

1. If calculation of a value has side-effects, and the value is needed several times, it can be stored into a local variable (or perhaps in the heap).

2. If an expression is used several times, it can be stored into a local variable to make the program shorter, and perhaps more efficient.

3. If an expression is too long, it can be split by using temporary variables for subexpressions.

It is simple to remove expressions so that only one form is needed. For example $v = m(e)$ can be changed to $t = e; v = m(t)$. If variable t is used only once, the temporary variable can be removed.

1.6.6 Sequences and substatements

An important notion in structured programming is a *substatement*. For example if one has statement

```
while (E) {
  S
}
```

it has substatements E and S. For refactoring, the important thing about substatements is that they can be moved into other parts of the program.

Sequences are rather simple structures. If there are nested sequences, they can almost always be removed. Correspondingly, if a new sequence is added, it does not need to have any nesting structure. An important thing to notice about sequences is their substatements. For example the sequence $a; b; c$ has substatements a, b, c, $a; b$ and $b; c$ that can be moved independently. The number of substatements depends on variable declarations.

Another concept that is similar to sequences are sets. For example a class can be thought to be a set of its members, because their order does not matter.

1.6.7 Lambda lifting

To get a flat function structure, one needs to invent a name for each lambda expression. All parameters used within these expressions must be passed to the new functions as arguments. The lambda expressions can be moved up and down in the function structure this way.

Many languages do not have this kind of functional structure, but for example Java has inner classes that have a similar purpose.

Because closure creation does not have side-effects, the lambda expressions can be moved up in command sequences, too. For the normal form, it would be most useful to have all functions on the top level, but for a user, it is probably best, if the functions are at the lowest possible level. For example, it would not make any sense that the arguments to a while-command would not be shown.

1.6.8 Generalization of functions

After the above considerations, each function in the program can be reduced into a sequence of form

$$\begin{aligned} t_1 &= f_1(v_{1,1}, \ldots, v_{1,n_1}) \\ \ldots & \\ t_m &= f_m(v_{m,1}, \ldots, v_{m,n_m}) \end{aligned}$$

The refactoring that remains is the generalization of functions. This means that a part of a function is changed into an argument for the function. We get a new normal form by cancelling all premature generalizations.

Generalization can be used to find common functionality. Another way to view generalization is to consider the cut-and-paste operation. For example, consider a case where one needs two classes c and c', which are otherwise equivalent, but in a method a in class c, there is a statement s that needs to be changed into s'. A natural solution would be to copy the class c, change its name to c' and then change the statement s into s'. This solution usually harms the readability and maintainability of the program. A better solution is to first generalize the class so that the statement s is given as a parameter to the class. In practice this specific case is usually done using the template method pattern.

1.6.9 Modules

For example lambda lifting produces new atoms into modules. For this reason it might be easier to handle the modules last. In language L, the notion of modules is very simple, and therefore the only refactoring that is needed, is moving members between modules. This refactoring is considered in detail in Chapter 4. Basically the problem is finding the most useful module structure.

We consider modules in more detail in Section 1.7.

1.7 Modularity and Objects

In Section 1.6, it was shown how parts of the programs that are related to design can be extracted into smaller parts or models. Out of these models, the focus of this dissertation is in the *module structure* of programs.

Modularity [75] is related to the idea that to solve a problem, it must first be divided into smaller subproblems. In software engineering, the problem to be solved is creating a program that fulfills its requirements. It can be thought that a requirement is already a description of what the program should do. When requirements are divided into smaller specifications, these specifications become more formal. More details can be added such as the

data needed to fulfill the requirements. In fact, often the most important requirements are about data that should be available.

A modular design approach has several advantages. It improves the maintainability of the programs. It makes reusing code easier, which in turn leads to better correctness and performance. It can improve flexibility, because for example a plug-in mechanism should be easy to implement.

The overall quality of software depends on the *architecture* of the software. One essential part of software architectures is to guarantee good modularity: All software architectures include descriptions of components (modules) and their relationships in the system (sometimes this is called the meta-architecture).

1.7.1 Modularity in programming languages

Several different modularity related constructs have been used in programming languages. Some of the most common ones are

- Methods, functions and procedures.
- Classes.
- Packages.

Many languages have also the ability to "nest" modules, for example inner classes and first-class functions.

In many languages, modules or classes can have *interfaces*, that describe the types of modules or classes.

Abstract data types are also based on the idea of refactoring: An abstract data type can be replaced with a similar one, and the semantics of the program stays the same.

There are three ways to use modules in programs:

1. In basic modularity, a program is just divided into smaller parts.
2. When a module is used via an interface, it can be replaced by another that uses the same interface. This is the basis for component based systems. The interfaces describe what kind of components the system has.
3. Finally, also relations between components are described explicitly. This can for example be implemented as a function that combines two components into one composite component.

All these ways have their own uses in the programs. These three uses of modules also show that the distinction between dynamic and static parts of the programs is not very clear.

1.7.2 Design heuristics for modularity

Three widely accepted measures to evaluate the modularity of software are the size of modules, the cohesion of modules, and the coupling between modules. In general, it is considered that modularity is good when there is low *coupling* between modules, and high *cohesion* inside each module [28].

In object oriented design, a class should represent a single logical concept, and not be a collection of miscellaneous features. Cohesion of a class tells how well it fulfills this design principle. Coupling on the other hand tells how strongly classes depend on other classes.

The idea of these heuristics is as follows: If a class is cohesive, the members of the module are related and it is useful that they can be found from the same module. If the size of a class is too large, it becomes harder to find the needed member in the module, and also understanding what is important in the module becomes harder. If a class has a lot of coupling, there are many related classes, and so the system becomes harder to understand.

The size is usually considered the most important software metric, and the same holds for modularity. If a module is too large, it is harder to understand and use. For example a wrong method can be selected if there are too many methods in one module. If the internal properties are not well documented, modifications inside the module can have unexpected consequences. On the other hand, if a module has too small size, it might seem disconnected from the other parts of the program.

1.7.3 Object-Oriented Terminology

The most popular languages today use the concept of classes instead of plain modules [68]. Therefore we need to consider what is the difference between these two concepts. It is important to realize that unlike dividing programs into modules, division into classes contains semantic information. In this sense the difference is like the difference between indentation and structured programming.

To understand, what object-oriented programming is about, we introduce a simple formalism that explains how object oriented concepts can be defined from more basic concepts. This is similar to defining structured statements in terms of higher order functions.

If there is a set of field names $\mathfrak{F}$, then the *object store* can be expressed as mapping $\mathfrak{o} : \mathfrak{I} \times \mathfrak{F} \to \mathfrak{V}$. The set $\mathfrak{I}$ is the set of *object identifiers* and the set $\mathfrak{V}$ is the set of possible *values*. It is said that each object has a unique identifier, and each object has one or several fields with values. Each object identifier is a possible value in itself, so we have $\mathfrak{I} \subseteq \mathfrak{V}$. Some languages also have other values, for example Java has machine integers and floating point values. These values are semantically different from object values.

Another mapping $\mathfrak{c} : \mathfrak{I} \to \mathfrak{C}$ gives for each object its class, where $\mathfrak{C}$ is the set of classes for a given software. Finally mapping $\mathfrak{r} : \mathfrak{C} \times \mathfrak{S} \to \mathfrak{M}$ resolves method signatures into actual methods, where $\mathfrak{M}$ is the set of methods and $\mathfrak{S}$ is the set of method signatures in the software. Now *method invocation* can be defined as

$$\mathfrak{o}.\mathfrak{s}(\mathfrak{v}) = \mathfrak{r}(\mathfrak{c}(\mathfrak{o}), \mathfrak{s})(\mathfrak{o}, \mathfrak{v})$$

This means that from object $\mathfrak{o}$, the method with signature $\mathfrak{s}$ is executed. In addition, value $\mathfrak{v}$ is given as an argument for the method. Here the object $\mathfrak{o} \in \mathfrak{I}$ is often called the *receiver* of the method.

To complete this simple semantic model, it would be necessary to associate methods with state transformations.

In addition to these concepts, there is the class hierarchy, which is defined differently for fields and methods.

Creation of a new object just returns a new object identifier, and then calls a *constructor function*. It seems that these concepts are almost inseparable.

Most complexity of object oriented programming comes from the class declaration construct. This construct can be thought to define one or more constructor functions. It can be thought that a class declaration first constructs a class lookup table, which is then shared by each object.

As a summary, a class declaration has the following semantic content.

- It declares mappings from objects to values. These mappings can only be applied to objects of this class, or to objects of its subclasses. This corresponds to the implementation where objects and fields are represented as integers.
- It declares a lookup table that contains functions. This table is added to the table that the inherited class has.
- A subset hierarchy is defined for the objects.

An important piece of terminology we need are *accessor methods*. There are two kinds of accessors: *getters* and *setters*. Getters are methods whose purpose is to return a value of the field, and setters are methods whose purpose is to change the value of a field.

1.7.4 Refactoring classes

As we saw, the fields can be interpreted as relations between objects and values. It is easy to use data structures such as maps instead of fields. This would however affect the performance negatively. Therefore, when moving fields, performance considerations must be taken into account. Luckily, it can usually be argued that the fields are in the correct place, when they

are associated with the same kind of object identifiers. There are several counter-examples, though:

- If a field has often a null value, it would seem that there is some kind of inefficiency. For example, the field might then represent a hot-key or voice command that could be used to activate graphical components, but most components would not have this kind of hot-keys.
- A piece of information might not be central to the intended meaning of the objects, instead it is auxiliary data for some kind of subsystem of the application.

For these reasons, it is common that instead of fields, more complex constructs are used to describe the properties of objects.

If a method is not called using dynamic binding, it is easy to move the method by converting it to a method that only operates on a single field or argument. The placing of methods is only related to the lookup tables.

1.7.5 Graphs

In the literature, metrics related to cohesion and coupling are defined in terms of graphs. Let $G = (V, E)$ be a graph with the set of *vertices* V and *edges* E. Then a metric $\mu : G \to \mathbb{R}$ calculates some property of the graph. To define metrics, it is useful to be able to restrict to some subset of graphs, $\mathcal{G}$ (*valid graphs*). Let $\approx$ be an equivalence relation, which tells when two graphs are *structurally similar*.

For example, a graph is *bipartite* if $V = A \cup B$ so that $E \subseteq A \times B$ and A and B are disjoint. Then the valid graphs could be the bipartite graphs, and two graphs with $V_1 = A_1 \cup B_1$ and $V_2 = A_2 \cup B_2$ could be structurally similar if their partitions are of the same sizes: $|A_1| = |A_2|$ and $|B_1| = |B_2|$.

Now one can define $[G] \in \mathcal{G}/\approx$ to be the equivalence class of $G \in \mathcal{G}$ with respect to relation $\approx$. Then the metric can be normalized into the interval $[0, 1]$ by

$$\mu^*(G) = \frac{\mu(G) - \inf \mu([G])}{\sup \mu([G]) - \inf \mu([G])}$$

.

For example, if the metric μ is the number of edges, and the valid graphs are bipartite graphs

$$\mu^*(G) = \frac{|E|}{|A||B|}$$

It is possible to define a metric for the number of pairs of vertices connected by a path with length n (α_n), or the number of paths with length n (β_n). For bipartite graphs, the metrics α_1 and β_1 are the same, and they count the number of edges. If n is even, we can have two variations: the

endpoints of the paths can be restricted to one of the sets A and B. We can notate $\alpha_n(G) = \alpha_{n,a}(G) + \alpha_{n,b}(G)$.

To take interfaces into account, each client has an interface through which it can use the services provided by the components. So we have a relation $R(a, b)$, that holds if a can use service b. The structural similarity is now calculated according to this relation. In this case it holds that

$$\alpha_n^* = \frac{\alpha_n(R)}{\alpha_n(E)}$$

For some purposes, it is useful to consider the graphs as probability spaces. For example if there is a graph with n edges, $P(a \wedge b) = \frac{1}{n}$, if $(a, b) \in E$, and $P(a \wedge b) = 0$ otherwise.

1.8 Program Analysis for Design

In this section we describe how program analysis can be used to assist in software design [56]. There are two levels: improving the software and improving the software environment.

Using automated analysis techniques, information can be gathered from a large body of software. This cannot practically be done otherwise. Classification of software entities is needed to gather more information about software. This information can then be used when evaluating software metrics or refactoring suggestion systems.

The limitation of this approach is that only the software is evaluated. What is missing is the history of how the development environment has been used.

1.8.1 Validation of metrics

The focus of this dissertation is in finding metrics that can be used to improve the quality of software. If one wants to use a software metric for quality improvement, there are several considerations:

- When can a software metric be applied in the development process?
- What kind of problems should be found with the metric?
- How can the problems that were found be fixed?

One reason why an internal cohesion metric should be useful, is that it can be applied to individual classes, unlike coupling metrics, which need the whole program to give reliable results. Because of this, such a metric can be used already at the beginning of software development. When a metric is used in a software development process, it should provide useful information

on the difference between the original and modified software versions. In an evolving design, adding new features should make the cohesion lower, and a refactoring should make it higher.

To evaluate the usefulness of a metric, one has to inspect the classes that the metric indicates having design problems. These problems are classified to true design problems and valid design decisions. If there seem to be enough true design problems, the impact of fixing the design problems can be evaluated. Also the effort needed for fixing the design problems should be evaluated. If it can be assumed that the impact of fixing the problems is larger than the effort needed for finding and fixing them, the metric can be used for software quality improvement.

Looking forward, if there is a refactoring suggestion that is supposed to implement a design improvement, one can evaluate it using design heuristics. If all design heuristics agree, then a reliable suggestion has been found. If design heuristics do not agree, then there are conflicting design issues. For a heuristic, it is possible to define a cost model that explains why this design heuristic really improves the software. If a cost model explains several different heuristics, then there is more evidence for the suitability of both the design heuristics and the cost model.

On the other hand, it may turn out that by using the metric, no design problems are found or there were too many false positives. Then one may attempt to classify the false positive findings, and define a new metric that does not generate these false positives. If there are still no true design problems found, a negative example of a design heuristic has been found, and the assumptions behind the heuristic have to be investigated.

1.8.2 Classifying Software Entities

To better understand the nature of software, the software entities can be classified. For example local variables can be classified by their roles [84]. This kind of classification helps in finding programming patterns. There are also many kinds of classifications for classes, see for example [67].

It is required from a good classification that it does not rely on the programming language, because the constructs in programming languages can be completely arbitrary. From this requirement it follows that the classifications might be complex to define formally. On the other hand the classification should be formally defined so that program analysis can be used to confirm the results.

A classification can be used to improve the design of programming languages by offering empirical input into language design. By classifying the software entities one can find programming patterns that can be a basis of new language features. On the other hand, it might turn out that some language features are not useful and should therefore be deprecated.

1.8.3 Discussion

As the size of software and the number of programmers working on software projects are increasing, measuring the software quality is gaining more importance. Measuring software quality using software metrics has two potential advantages. First, it can help to predict, how a software process will succeed, and second, it can help to improve the quality of software.

Code reviews are currently used to improve the quality of software. Software metrics automate part of this process.

The quality of software is weakened by design problems. These problems are caused by program evolution or by errors made in the design phase. They can make programs error prone, harder to understand and maintain. At least two characterizations of design problems exist in the literature. Anti-patterns [19] are modeled after design patterns. Code smells are based on refactorings [43]. Both kinds of problems can be detected using software metrics, as is discussed in [56]. Software metrics and refactorings have a close relationship; metrics are used to find problems that need refactoring, and the values of metrics should improve as a result of refactoring.

1.9 Implementation

Implementing advanced software product metrics for Java includes several challenges. The most fundamental one is that global analysis is required for finding and analyzing semantic properties of programs. For this reason, the analysis needs to be implemented very efficiently, or it does not scale to realistic, very large programs. To improve the metrics, more complex analyses such as points-to analysis are needed. Also the definitions of the metrics can be complex and the metrics can have several different variations. Further, Java also has a rather complicated syntax, and the analysis cannot be done properly until the relevant information has been extracted from the program.

To overcome these challenges, a framework for efficient software analysis and metrics calculation is proposed in the present work. This framework has been implemented as a three-part tool, called QuickMetrics. The first part is an interface with integrated development environment (IDE). This component converts the program files into XML files. These files are stored in the second part, in an XML database. Using a query language XQuery, the database can be searched, and features needed in software analysis can be extracted into tables. These tables can then be processed efficiently using a BDD based language for relational computations.

User-defined analyses and metric calculations can be performed using a two-step approach, where the needed information is first extracted from the program using XQuery, and a semantic analysis is then performed using

a BDD based analysis step on the relations returned by the query. This two-step approach comprises the third part of our framework.

1.9.1 Integration with Eclipse IDE

Different software development tools are usually utilized in IDEs. Current IDEs, such as Eclipse, provide several useful features to help in programming, like syntax and type checking in the program editors, and tools for refactoring. As a bridge between the software analysis and the development environment, the framework of the present work includes a plug-in for Eclipse IDE, which converts the internal representation of Eclipse to XML, and vice verse.

The Eclipse plug-in has two parts: an importer and an exporter. The exporter stores program files into an XML database. This can be done when the program is compiled to keep the database up-to-date. In general, the exporter has two parts: parsing and type checking. Both parsing and type checking can be done using the Eclipse JDT library. Type and binding information is often needed by analysis, so it is very expedient to store it into the database.

To make implementation of the analyses easier, the elements stored in the database must be clearly identified. Top-level types are identified by their fully qualified names. Type members are identified by the class identifier and their signature. Expressions and statements are identified by their file name and location.

The importer can read files from a database. If also comments and line numbers are stored into the database, no information is lost when transforming into XML and back into text format. In theory, it is possible to completely remove the text based representation, and store programs only into the XML database.

Another feature that is useful for IDE integration is getting information for an element of a program. For example, if an analysis produces a relation for expressions, the related elements can be easily queried. Analysis relations are always assumed to be up-to-date. Actually, these relations can be computed on demand.

It would be possible to store byte codes into a database instead of the source code. The advantage of this would be that bytecode has less linguistic constructs. The disadvantage of this solution is that the developers are much more familiar with the structure of the source code than the structure of byte code.

1.9.2 XQuery database

After the program has been stored into an XML database, it is possible to use XQuery to extract the information needed by the software analyses. The database implementation used was pathfinder [81].

Examples of program query systems are magellan [33] and codequest [47]. Like magellan, we use XQuery for querying the code database. However in magellan there is no easy way to define efficient analyses. Codequest is more efficient, because it uses a relational back end. Some analyses are easier to implement using BDDs because they use very large tables.

The code database can be queried in two different ways. Queries that are useful for software analyses are typically applied to all compilation units, type declarations, etc. Other kind of queries can be used for searching code. Searching is more efficient than traditionally in IDEs, as programs are stored in a database instead of plain text format. It is also possible to do structural search, and what is still more important, use the included type and binding information.

In general, the queries are easy to write, and efficient to implement. However, sometimes the syntax of Java is rather complex to interpret, for example qualified names may cause problems. For this reason, user-defined functions can be added to make the writing of queries easier.

Figure 1.1 shows a query that is needed in a points-to analysis. This query returns all object creation sites. It also returns the class of the created objects. The value of the variable $\$prog$ is the whole program. Tag *new* is for object creation expressions. Path expression *type* / *key* returns the type of the created object, and then returns the unique identifier for the class. This identifier is calculated by the compiler when the program is converted into XML. The attribute *exprid* is a unique identifier for each expression in the program code.

```
for $cons in $prog//new
where data($cons/type/key) ≠""
return <p><key>{data($cons/type/key)}</key>
    <id>{data($cons/@exprid)}</id></p>
```

Figure 1.1: An example query which returns the object creation expressions with type information.

Because the queries can be calculated separately for each compilation, it is simple to implement incremental computation efficiently. A driver executes the query for each updated compilation unit. The query is executed for both old and new versions of the unit. To capture changes, we just compare these results with the old results. The change of a relation is represented as

the tuples that have been added to the relation, and the tuples that have been removed from the relation.

1.9.3 Relational language

With a relational language, several useful analyses can be defined. In principle, XQuery can be used to express relational operations. In practise, this is however too slow for handling recursive queries. A BDD based implementation is more efficient, especially when relations might have a lot of redundant information, as is often the case in program analysis.

We have implemented a new BDD based language HD-BDD. Two existing examples of BDD based languages are bddbddb [91] and CrocoPat [9]. HD-BDD is similar to these languages. Because the syntaxes of bddbddb and crocopat are based on logic programming, they do not have a natural support for arithmetic operations, and they are thus not well suited for metrics calculation. Following the tradition of database query languages, HD-BDD has good support for metrics.

HD-BDD is a simple functional language with XQuery-like syntax, see Figure 1.2. HD-BDD has support for three kinds of data types: atoms, relations and numeric values. Relations are defined between atoms or numeric values. They are internally represented as BDDs. Atoms belong to finite domains. Domains and relations are loaded into HD-BDD from XML files created by database queries.

The language has the following basic operations for relations:

- Intersection of relations,
- Union of relations, and
- For-Where-Return-expression (FWR), see below.

In addition, HD-BDD includes let-binding and mutually recursive definitions.

The FWR-expression corresponds to a natural join in relational algebra, and it is similar to the for-expression in XQuery. The semantics of the FWR-expression is a set comprehension. If we have expression

$$\begin{array}{l}\textbf{for } (v_{1,1},\ldots,v_{1,n_1}) \textbf{ in } e_1,\ldots,(v_{m,1},\ldots,v_{m,n_m}) \textbf{ in } e_m\\ \textbf{where } v_{f(1)} = v_{g(1)} \textbf{ and}\ldots\textbf{and } v_{f(l)} = v_{g(l)}\\ \textbf{return } (v_{r(1)},\ldots,v_{r(k)})\end{array}$$

and expression e_i evaluates to relation R_i, the relation expressed by the FWR-expression is

$$\begin{array}{c}\{(v_{r(1)},\ldots,v_{r(k)}) \mid (v_{1,1},\ldots,v_{1,n_1}) \in R_1,\ldots,\\ (v_{m,1},\ldots,v_{m,n_m}) \in R_m,\\ v_{f(1)} = v_{g(1)},\ldots,v_{f(l)} = v_{g(l)}\}\end{array}$$

$$
\begin{array}{rcl}
e & ::= & \textbf{empty} \\
& | & v \\
& | & e \cap e \\
& | & e \cup e \\
& | & e - e \\
& | & \textbf{for } (v, \ldots, v) \textbf{ in } e, \ldots, (v, \ldots, v) \textbf{ in } e \\
& & \textbf{where } v = v \textbf{ and} \ldots \textbf{and } v = v \\
& & \textbf{return } (v, \ldots, v) \\
& | & \textbf{let } v \ = \ e \ e \\
& | & \textbf{let rec } v \ = \ e \textbf{ and} \ldots \textbf{and } v = e \ e
\end{array}
$$

Figure 1.2: The syntax of HD-BDD.

The value of recursive definitions is defined to be the fixpoint, when the definition is repeatedly applied to an empty set. Combination of FWR-expressions and recursive definitions is very similar to Datalog like BDD-based languages. The implementation also supports function definitions. Recursion can be used freely, and the language has first class functions with lexical scoping. Relational calculation can be implemented in the usual way using for example trees, or it can be implemented using BDDs. BDDs are hard to optimize because their performance can be unpredictable.

Programs written in HD-BDD are interpreted, because most of the time is spent doing the relational calculations.

1.9.4 Example

As an example analysis using HD-BDD, let us calculate the transitive closure of call trees. For an analysis of a simple call tree, one needs a table that contains what methods call each other ($\$call$), and a table which tells which methods overload some another method ($\$over$).

Table $\$call\_over$ is formed as follows:

$$
\begin{array}{l}
\textbf{let } \$call\_over := \\
\quad \textbf{for } (\$caller, \$called) \textbf{ in } \$call, \ (\$sub, \$super) \textbf{ in } \$over \\
\quad \textbf{where } \$called = \$super \\
\quad \textbf{return } (\$caller, \$sub)
\end{array}
$$

The table $\$call\_over$ resolves the dynamic binding of method calls. Every possible method, that overloads the target method, might be called. Then

the transitive closure in the definition of $\$call_tr$ is calculated:

$$
\begin{array}{l}
\textbf{let rec } \$call\_tr := \$call\_over \cup \\
\quad \textbf{for } (\$caller, \$called) \textbf{ in } \$call\_over, \\
\quad\quad (\$caller1, \$called1) \textbf{ in } \$call\_tr \\
\quad \textbf{where } \$called = \$caller1 \\
\quad \textbf{return } (\$caller, \$called1)
\end{array}
$$

The results of the analysis above can be improved using a points-to analysis. Assume that the results of the points-to analysis are in a table *points* where $(e, c) \in points$ if the value of expression e can be of class c, and relation $call(te, m)$ states that for the expression te the method m will be applied. Table *resolve* has element (c, m, m'), if the method signature of m applied to class c would actually call method m'. A table that tells if a method contains an expression is also needed.

Using points-to analysis, overloading of methods can be resolved as:

$$
\begin{array}{l}
\textbf{let } \$call\_over := \\
\quad \textbf{for } (\$e, \$c) \textbf{ in } \$points, \ (\$te, \$m) \textbf{ in } \$call, \\
\quad\quad (\$m2, \$me) \textbf{ in } \$methods, \\
\quad\quad (\$mc, \$m3, \$m4) \textbf{ in } \$resolve \\
\quad \textbf{where } \$e = \$te \textbf{ and } \$m3 = \$m \\
\quad \textbf{and } \$me = \$te \textbf{ and } \$c = \$mc \\
\quad \textbf{return } (\$m2, \$m4)
\end{array}
$$

Method m_2 has an expression, which calls a method with the signature m. In this call, the target can be of class c. As a result, m_2 can call the method from c with signature m.

1.10 Description of the thesis work

Now that we have described the main ideas behind software product metrics and related areas, we concentrate on cohesion metrics for software quality improvement. Cohesion seems to be the property of modularity that is most hard to capture by metrics.

In Chapter 2, common cohesion metrics are described. It is then evaluated how these metrics can be used as design heuristics. It will turn out that LCOM like metrics are useless as design heuristics. The reason for this is that there are some false assumptions in the common definition of cohesion.

Another interpretation of cohesion is presented in Chapter 3. Now external relationships between class members are considered instead of internal ones. A metric called LCIC is defined based on this interpretation. LCIC is evaluated in the same way as LCOM. Also design patterns and refactorings

are considered in this chapter. It is found out that this metric is much better suited as a design heuristic than LCOM.

In Chapter 4, implementation of a refactoring suggestion system for modularity is proposed. This is done by defining a metric that measures several modularity related aspects: size, cohesion, coupling and significance. It is found out that the quality of the suggestions produced by the metric is good, and the interpretation of the modularity implicit in our metric is close to the intuition of programmers.

Chapter 2

Internal Cohesion Metrics

2.1 Introduction

In this chapter the *internal cohesion* of software modules is discussed. This means that the aim is to find out whether a module is cohesive by inspecting it separately from other parts of the system. *External cohesion* is studied in Chapter 3. There, the degree of cohesion is determined by checking how the module is used by the other parts of the system.

The first task in this chapter is to define what cohesion, and especially its internal interpretation, means. To make the concept of internal cohesion relevant for the analysis of software quality, one has to show how the principle of high internal cohesion is related to object-oriented design principles. It is important to realize that cohesion is related to the *modularity* of software ([28]), and it may be in conflict with object-oriented principles.

For empirical investigation, we propose a family of internal cohesion metrics for the Java programming language, and then measure these metrics for the classes of selected open-source projects. Before investigating the relation of the metrics to software quality, their statistical properties are investigated. The most important of these properties is the relation of internal cohesion metrics to other metrics. The most basic software metric is the size of software artifacts. It will be shown that internal cohesion is strictly related to the size of the classes and that it is not simple to separate cohesion from size.

We compare the results between different metrics variations, and try to find out what kind of properties these variations measure, and what are their differences. For metrics that give similar results, we evaluate, which one is the most natural.

To use a single metric as a design indicator, there needs to exists a threshold value t such that there is a high probability for the existence of a design error if the metric value of a component is larger than t. To evaluate

the metric, it is necessary to inspect the classes and determine whether the classes have design problems. Because the cohesion metrics measure the classes as whole, and the size of some classes is large, it is not always easy to pinpoint what exactly causes the problem in the class. Finding the specific problem is not the goal of internal cohesion metrics for classes, rather just finding out which classes have cohesion problems. However, if the metrics are to be used for quality improvement, eventually the problem needs to be found.

The practical tests indicate that internal cohesion cannot be used for software quality improvement. Because of this, it is necessary to evaluate why the object oriented design principles are not useful when trying to improve the internal cohesion. To better understand what internal cohesion means, a classification for the relationships between methods and fields in classes is proposed.

The most important reason why internal cohesion metrics cannot be used as reliable quality indicators is that methods do not, and cannot usually encapsulate the fields of an object.

2.1.1 Outline of the research

The outline of this chapter is as follows. A general definition of an LCOM-style cohesion metric is first given. With this definition, it is possible to define internal cohesion metrics by giving different interpretations for graph cohesion and elements of classes. Exact definitions of the metric variations and attributes are given in Section 2.3. This formulation is based on a framework that is similar to that of Briand et al.[15] The definitions are general enough to be applied to any class-based language.

The most natural way to define cohesion is to include inherited methods and fields into the calculation of the metrics. This most closely corresponds to the semantic idea of object-oriented programming. The *flattened LCOM* that implements this kind of metric calculation is studied in Section 2.4. Flattened LCOM can be split into three pieces in a natural way: the cohesion of parent, cohesion of a locally defined part, and connection between inherited part and locally defined parts. In Section 2.5 we evaluate a local version of LCOM, where inheritance is not taken into account. This corresponds to classes as software artifacts. The connections are studied in Section 2.6. The most basic way to understand the lack of internal cohesion is to calculate the most coarse-grained cohesion metrics. This case is considered in Section 2.7. In Section 2.8, the metric LCC, TCC and dual TCC are considered. The results of these metrics are between the results from counting the number of components and LCOM. Finally in Section 2.9, the reasons for low cohesion will be classified.

This chapter is partially based on publications [58], [59] and [63].

2.2 Internal cohesion

To define the concept of cohesion, one first has to define what is meant by a module, since cohesion is a property of modules. In general, a *module* is just a collection of smaller software artifacts. In object oriented programming, classes have the role of modules: A *class* is a collection of *members*, that is, methods and instance variables. However, as was pointed out in Section 1.7, the classes are not simple collections: they include semantic content related to late binding of methods.

If a *module is cohesive*, the members in the module are similar or related. Obviously two atoms can be more or less related or similar, so a *cohesion metric* is based on a more fundamental metric of the strength of relationship between methods.

As an example of similarity, one can check whether two methods call the same methods. In this case, if two atoms are very similar, the similar component should be refactored out. Another possibility would be checking if two methods use the same types.

In most proposed measures, relationships are preferred in cohesion measurement over similarity. When looking at internal cohesion, the relationships that are visible by inspecting the module or class are measured. Basically this means that the relationships we are interested in are methods calling other methods or accessing fields.

2.2.1 Motivation for Internal Cohesion

Object oriented design (OOD) principles are closely related to the concept of internal cohesion for classes. In OOD, the first task is to recognize the objects. The attributes of objects and relations between objects are then found. In the implementation, the relations between objects are usually translated into attributes. A class is supposed to *encapsulate* data by providing high-level operations on that data. The data are considered as low-level details that should be hidden.

Associating data with a class is a semantic concept, and therefore not purely a design concept. This semantic concept is tied to classes as a feature of object oriented languages. Because it is a language feature, it always overrides design considerations. The class construct can be compared to branching constructs in structured programming. Both structured programming and object-oriented programming added semantic constructs to programming languages to address design issues. The question now is, should the semantic concept of classes lead to good internal cohesion.

The key point is that also operations are associated with classes. When more operations are added for a class, there should be more internal relations between the fields, and therefore the internal cohesion of the class should

become higher. A class is cohesive, if the operations are really related to each other and the data of the class.

According to the above reasoning, internal cohesion should be a good way to measure the quality of object oriented programs. However, it will be seen that the internal cohesion is too rarely high, which makes it impossible to consider internal cohesion a reliable measure of program quality.

There are two threats for internal cohesion: (i) the fields associated with an object are not necessarily related, and (ii) the operations might not be related to objects, but to more complex entities.

One known problem for cohesion measurement is due to the accessor methods. Accessor methods can be thought to be breaking encapsulation, and therefore bad design. However accessor methods are used so often that they are usually ignored when calculating cohesion. Accessors can be seen as an indication, that many methods should not really associated be with classes. Object oriented languages also have features, such as friends in C++, that are meant for more complex understanding of encapsulation.

2.2.2 Previous research in internal cohesion

The first metric to measure internal cohesion in object oriented languages was Lack of Cohesion On Methods (LCOM), which was introduced by Chidamber and Kemerer [25, 26]. This metric is based on the assumption that if methods and instance variables of a class are interconnected, then the class is cohesive. Henderson-Sellers [49] defined the metric LCOM* that is similar to LCOM, but has a fixed scale of values. Briand et al. [15] observed that the scale of LCOM* is from 0 to 2, and gave a refined version of this metric with scale from 0 to 1. Other similar cohesion metrics are LCC (Loose Class Cohesion) and TCC (Tight Class Cohesion) [11], and CBMC [22].

In addition to these metrics, there is another way to measure cohesion as proposed by Briand et al [17, 18]. This metric is based on the interfaces of the classes, ie. what types are used there. Surveys of cohesion metrics have been made by Briand et al. [15] and Chae et al. [22].

It is characteristic of the research on cohesion metrics that it seems to be unclear what kind of problems the metrics are expected to find, and for which kind of programs the metrics give different results. Because of this it is hard to see which are the advantages and disadvantages in using each metric. In addition to this problem, applying metrics effectively is hard because the cohesion values measured by the metrics have hidden dependencies on other attributes. An example of this is that the metrics tend to give lower cohesion for big classes. For example the correlation between the number of local variables and traditionally defined LCOM* is over 0.5. Finally there is not much empirical validation for the usability of cohesion metrics. Because of these problems, cohesion metrics are currently not in wide-spread use.

2.3 Defining cohesion metrics and attributes

Several metrics have been proposed to measure cohesion of object-oriented programs. Following the original LCOM metrics [25, 26, 49], most object-oriented cohesion metrics, as discussed in [11, 10, 21, 37, 23, 22, 40, 8, 34, 5, 29, 64, 24], define a cohesion graph that relates the fields of the class to the methods of the class. The metric then measures for example the sparseness or the connectedness of this graph.

A cohesion measure is usually applied to a graph that has methods and instance variables as nodes and relations describing variable usage as edges. Measuring cohesion can be varied in five different ways.

1. What is the measured structure: classes, packages, modules?
2. What are the properties of these structures that need to be cohesive?
3. What are the users of these properties?
4. What is the relation between properties and their users?
5. How is the cohesion measured based on this relation?

For the purposes of this chapter, the measured structures are classes, the properties are the fields, and the users are the methods.

Based on these observations, we give a formal definition of the cohesion metric framework. First, it is assumed that there are sets for classes $\mathfrak{C}$, methods $\mathfrak{M}$ and instance variables or fields $\mathfrak{F}$. To define the above variations modularly, a *cohesion metric* is defined to consist of four parts:

1. Relation Meth $\subseteq \mathfrak{M} \times \mathfrak{C}$ between methods and classes that should be used in the measurement.
2. Relation Var $\subseteq \mathfrak{F} \times \mathfrak{C}$ between instance variables and classes that should be used in the measurement.
3. Mapping Use : $\mathfrak{C} \to \mathcal{G}$ that returns for each class a graph describing when a method uses a variable.
4. Evaluation function for bipartite graphs $\mu : \mathcal{G} \to [0, 1]$. Value 0 is interpreted as the best cohesion value, and 1 denotes the worst cohesion. The measured graphs are bipartite, because they have only relationships between methods and fields, and not for instance between two methods.

For relations, we use the notation $R(a) = \{b \mid b\ R\ a\}$.

To calculate the result of the measure for a class $\mathfrak{c} \in \mathfrak{C}$, we first represent the nodes of the bipartite graph as two sets Meth($\mathfrak{c}$) and Var($\mathfrak{c}$). In this graph

the edges are between these two sets: $\{(\mathfrak{m}, \mathfrak{f}) \in \text{Use}(\mathfrak{c}) \mid \mathfrak{m} \in \text{Meth}(\mathfrak{c}), \mathfrak{f} \in \text{Var}(\mathfrak{c})\}$. This graph is called the *cohesion graph* and the metric is calculated from this graph. The usage relation is not enough to represent the cohesion graph, because there would be no way to represent a node without any connections with this relation.

2.3.1 Model for programs

To make measurements of properties of programs, it is useful to first define a model of programs as described in the introduction. This model can then be used to give a precise meaning to different variations of the cohesion metric.

The basic *locations* in the data flow and control flow of the program are classes $\mathfrak{C}$, methods $\mathfrak{M}$ and fields $\mathfrak{F}$. Class $\mathfrak{c} \in \mathfrak{C}$ has a set of locally defined methods $\text{meths}(\mathfrak{c}) \subseteq \mathfrak{M}$ and a set of locally defined instance variables $\text{vars}(\mathfrak{c}) \subseteq \mathfrak{F}$. As an additional structure, there is a set $\text{parents}(\mathfrak{c}) \subseteq \mathfrak{C}$ which is the set of direct parent classes. The reflexive transitive closure $\text{parents}^*(\mathfrak{c})$ is then the set of all parent classes of $\mathfrak{c}$. The set of subclasses for a class $\mathfrak{c}$ can be defined as

$$\text{subclasses}(\mathfrak{c}) = \{\mathfrak{c}' \in \mathfrak{C} \mid \mathfrak{c} \in \text{parents}^*(\mathfrak{c}')\}$$

A *virtual method* is a method which can be invoked using dynamic binding. A *virtual method call* is a method call, which is dynamically bound. A virtual method can also be called statically, for example when a method calls a method of the base class using `super`-syntax. Virtual method calls are described using *method signatures* $\mathfrak{S}$. These signatures are identifiers that can be used to determine which method is invoked when making a virtual method call to an object that is an instance of a known class. For example in Java this could be the method name and the types of the arguments. The set $\text{resolve}(\mathfrak{s}, \mathfrak{c})$ is a singleton set that contains the method with signature $\mathfrak{s}$ in class $\mathfrak{c}$, and it is the empty set if the class $\mathfrak{c}$ has no such method. We extend in the present study this notation for sets of signatures as $\text{resolve}(\mathfrak{S}, \mathfrak{c}) = \bigcup_{\mathfrak{s} \in \mathfrak{S}} \text{resolve}(\mathfrak{s}, \mathfrak{c})$.

Note that all sets defined above can be calculated statically: From the static type of the variable that a method is called from, one can see whether the method is static or not, and based on the signature, it is possible to find a corresponding method from the class. What cannot be determined statically is the dynamic type of an object, but this is not needed in the present analysis. Instead it is assumed that the object referred by variable `this` has some specific class, and the calculations are made based on that.

We also define sets of private and static variables $\mathfrak{F}_{\text{priv}}$ and $\mathfrak{F}_{\text{sta}}$, and sets of private, static and abstract methods $\mathfrak{M}_{\text{priv}}$, $\mathfrak{M}_{\text{sta}}$ and $\mathfrak{M}_{\text{abs}}$. The private fields and methods are methods that can only be called inside the classes.

Other methods and fields are called public. Abstract methods are methods that have only signatures and not an implementation. The set

$$\text{abstsig}(\mathfrak{c}) = \{\mathfrak{s} \in \mathfrak{S} \mid \text{resolve}(\mathfrak{s}, \mathfrak{c}) = \{\mathfrak{m}\}, \mathfrak{m} \in \mathfrak{M}_{\text{abs}}\}$$

is the set of abstract method signatures for a class $\mathfrak{c}$.

2.3.2 Usage relation

It seems easy to define the relation for usage. One could just have an edge between a method and the instance variables that it accesses, and other methods it calls. But it might be wise to make the graph bipartite, because the methods are compared wrt. instance variables in most definitions of cohesion metrics. And when *indirect usage* is considered, there might be a need for looking this-variables passed as arguments to some methods.

As described in Section 1.5, it is convenient to define language $A \subseteq \Sigma^*$ to describe the usage relationships. The set $U \subseteq \Sigma \times \Sigma$ describes how a method uses other locations in the program directly. Given U, it is simple to define A.

To properly define the usage relation, it is necessary to consider which syntactic forms the accesses of methods and fields might have. They can be classified as:

1. access($\mathfrak{f}, \mathfrak{e}$), where $\mathfrak{f}$ is a field and $\mathfrak{e}$ is some expression where the field is accessed. The type of the expression $\mathfrak{e}$ is type($\mathfrak{e}$). Here it is assumed that type is the set of subclasses of the corresponding static type.
2. access($\mathfrak{f}$, **this**) is the access of field from this variable.
3. access($\mathfrak{m}, \mathfrak{e}_1, \ldots, \mathfrak{e}_i$), where $\mathfrak{m}$ is a method signature and $\mathfrak{e}_i$ are expressions. The method call can be static.
4. access($\mathfrak{m}$, **this**, $\mathfrak{e}_1, \ldots, \mathfrak{e}_i$) is a method call from this-variable.
5. access($\mathfrak{m}$, **super**, $\mathfrak{e}_1, \ldots, \mathfrak{e}_i$) is a method call from super-variable.

Probably the most useful formulation of the usage relation is to ignore this-passing and just have as locations the methods and fields with associated classes. Then we have

$$\mathfrak{m}(\mathfrak{c}), \mathfrak{f}(\mathfrak{c}) \in \Sigma$$

That is, the set Σ of locations in the program consists of methods $\mathfrak{m}$ and fields $\mathfrak{f}$ associated with classes $\mathfrak{c}$. For each class, it may be assumed that the associated methods are the methods that are defined in the class or its superclasses.

The usage relation U consists of pairs $\mathfrak{l} \mapsto \mathfrak{l}'$, meaning that the location $\mathfrak{l}$ uses $\mathfrak{l}'$. Relation U can thus be defined by inspecting each location $\mathfrak{m}(\mathfrak{c})$ as follows:

- for field access, access($\mathfrak{f}, \mathfrak{e}$): $\mathfrak{m}(\mathfrak{c}) \mapsto \mathfrak{f}(\text{type}(\mathfrak{e})) \in U$;
- for field access, access($\mathfrak{f}$, **this**): $\mathfrak{m}(\mathfrak{c}) \mapsto \mathfrak{f}(\mathfrak{c}) \in U$;
- for method access, access($\mathfrak{m}'$, **this**, $\mathfrak{e}_1, \ldots, \mathfrak{e}_n$): $\mathfrak{m}(\mathfrak{c}) \mapsto \text{resolve}(\mathfrak{m}', \mathfrak{c})(\mathfrak{c}) \in U$;
- for a general method access, access($\mathfrak{m}', \mathfrak{e}, \mathfrak{e}_1, \ldots, \mathfrak{e}_n$): $\mathfrak{m}(\mathfrak{c}) \mapsto \text{resolve}(\mathfrak{m}', \text{type}(\mathfrak{e}))(\mathfrak{c}) \in U$.

Then for the following sample case

```
class K {
  private K up, down;
  public void connectUp(K a) {
    this.up = a;
    a.connectDown(this);
  }
  public void connectDown(K a) {
    this.down = a;
    a.connectUp(this);
  }
}
```

both methods would be seen to access both fields.

As was said above, given relation U one can define language A to be the smallest set such that

- $\mathfrak{l} \in A$ for each $\mathfrak{l} \in \Sigma$.
- If $w \cdot \mathfrak{l} \in A$, then $w \cdot \mathfrak{l} \cdot \mathfrak{l}' \in A$, if $(\mathfrak{l}, \mathfrak{l}') \in U$.

We say that language B is *valid* or satisfiable if

$$\text{valid}(B) = A \cap B \neq \emptyset$$

Indirect usage of methods in a class can be defined in two different ways: First, only internal calls to `this` could be included. To implement this we would drop items 1 and 4 from the definition of U. The second possibility is to keep the original definition of U and use relation

$$\text{uses}^*(\mathfrak{c}, \mathfrak{m}_1, \mathfrak{m}_2) = \text{valid}(\mathfrak{m}_1(\mathfrak{c}) \cdot \mathfrak{M}(\mathfrak{c})^* \cdot \mathfrak{m}_2(\mathfrak{c}))$$

The operator $\text{uses}^*(\mathfrak{c}, \mathfrak{m}_1, \mathfrak{m}_2)$ returns all the instance variables and abstract method signatures that are used by the method $\mathfrak{m}_1$, when recursively calling the target object. If another type of object is called, then what is used in that call is ignored.

Other variations of the uses* operator could be ignoring indirect calculation completely, or restricting indirection to for example public, private or locally defined methods.

There are issues related to inner classes that should be addressed here. An inner class might be defined inside a method, and this inner class might use the instance variables of the outer class. In this case all the variables used by the inner class are added to the variables used by the method. For inner classes, there is still one question, that is how to measure cohesion of the non-static inner classes, that have been defined as members of classes. Perhaps the best way would be to handle them like methods in the measurement of the outer class, and measure the inner classes similarly to inherited classes. This question is omitted in the present study.

It might be possible to count non-static inner classes as methods in the outer class. Also, if a method defines an inner class, it is a good idea to include the instance variables that might be used in that inner class to the set of variables used by the method. Changing the implementation to take this into account makes the cohesion higher. Note that this would be problematic in dynamic methods for calculating the cohesion.

2.3.3 Fine-grained usage relation

Above, the locations of the program were methods associated with possible types of this-variables. Instead of that, all argument types could be associated with the methods in the same way. Methods with resolved types are considered to be locations. They are notated $\mathfrak{m}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \in \Sigma$, where $\mathfrak{c}_1$ stands for the type of the receiver and $\mathfrak{c}_2$ to $\mathfrak{c}_n$ describe the types of the arguments. Other locations are of form $\mathfrak{f}(\mathfrak{c}) \in \Sigma$ for accessing field $\mathfrak{f}$ from class $\mathfrak{c}$.

For each expression $\mathfrak{e}$ in method $\mathfrak{m}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)$ the possible types are determined as follows:

- If expression $\mathfrak{e}$ is `this`, then $\text{type}(\mathfrak{e}) = \mathfrak{c}_1$.
- If $\mathfrak{e}$ is the i'th argument of the method, then $\text{type}(\mathfrak{e}) = \mathfrak{c}_i$.
- Otherwise, if $\mathfrak{e}$ has type $\mathfrak{c}$, then $\text{type}(\mathfrak{e}) = \text{subclasses}(\mathfrak{c})$.

Then, the method and field access can be defined as follows: For field access $\text{access}(\mathfrak{f}, \mathfrak{e})$, we get

$$\mathfrak{m}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \mapsto \mathfrak{f}(\text{type}(\mathfrak{e})) \in U$$

and for method access $\text{access}(\mathfrak{m}, \mathfrak{e}_1, \ldots, \mathfrak{e}_n)$, we get

$$\mathfrak{m}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \mapsto \text{resolve}(\mathfrak{m}, \mathfrak{c})(\text{type}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)) \in U$$

if $\mathfrak{c} = \text{type}(\mathfrak{e}_1)$.

2.3.4 Handling this-passing

As an extended use relation, uses$'(\mathfrak{m}, \mathfrak{c})$ is defined to be the set of variables that first of all includes uses$^*(\mathfrak{m}, \mathfrak{c})$. In addition, if `this`-variable is given as an argument to a static method call, the set includes the instance variables or methods of the argument objects that are used directly or indirectly, and adds these to the result set.

There are three variations to this scheme: (i) This-passing into virtual methods can be ignored, (ii) all known possibilities are combined, or (iii) it can be assumed that a virtual method uses all members of its arguments. From now on, we use the alternative (i).

There are two possible *call styles*: focused argument **arg**, or normal call **normal**. Suppose that there are locations $\mathfrak{m}(i, (s, \mathfrak{c})) \in \Sigma$, where s is the style, i is the index number of the focused argument, and $\mathfrak{c}$ is the type of the focused argument.

For each expression $\mathfrak{e}$ in method $\mathfrak{m}_i(\mathfrak{c})$ we determine the possible types as follows:

- If expression $\mathfrak{e}$ is `this` and $i = 0$, then type$(\mathfrak{e}) = (\mathbf{arg}, \mathfrak{c})$.
- If $\mathfrak{e}$ is argument i, then type$(\mathfrak{e}) = (\mathbf{arg}, \mathfrak{c})$.
- Otherwise, if $\mathfrak{e}$ has type $\mathfrak{c}'$, then type$(\mathfrak{e}) = (\mathbf{normal}, \text{subclasses}(\mathfrak{c}'))$.

Then the method and field access are defined as follows: For fields access$(\mathfrak{f}, \mathfrak{e})$, we get

$$\mathfrak{m}(i, \mathfrak{c}) \mapsto \mathfrak{f}(\text{type}(\mathfrak{e})) \in U$$

and for methods access$(\mathfrak{m}, \mathfrak{e}_1, \ldots, \mathfrak{e}_n)$,

$$\mathfrak{m}(i, c) \mapsto \text{resolve}(\mathfrak{m}, \mathfrak{c}_0)_j(\text{type}(\mathfrak{e}_j)) \in U$$

where $\mathfrak{c}_0 = \text{type}(\mathfrak{e}_1)$.

A chain of this-calls is now $\mathfrak{M}(0, \mathbf{arg})^*$. This-passing is represented as $\mathfrak{M}(*, \mathbf{arg})$. Focused argument is passed forward if $\mathfrak{M}(*, \mathbf{arg})$. Focused argument is called again as $\mathfrak{M}(0, \mathbf{arg})$. This-passing chain is then $\mathfrak{M}(*, \mathbf{arg})^*$. So the combination of this-chain and this-passing chain is just $\mathfrak{M}(\mathbf{arg})^*$.

A path for $\mathfrak{m}$ accessing $\mathfrak{f}$ in class $\mathfrak{c}$ is then

$$\mathfrak{m}(\mathfrak{c}) \cdot \mathfrak{M}(\mathbf{arg})^* \cdot \mathfrak{f}(\mathbf{arg})$$

2.3.5 Variables

The second issue that needs to be considered when discussing cohesion metrics is the concept of fields or instance variables of classes. The most important source of variation is inheritance. It is not entirely clear whether

measuring inheritance should actually be called cohesion and not the closely related concept of coupling. There are two ways to think about inheritance, first is that the child class is a kind of module that uses the parent class to implement a new class, and in the second way the child class is a kind of module where the features of the parent class are enriched. It is our opinion that the second way is closer to object oriented thinking, where there are is-a-relations that describe inheritance, and uses-a-relations for composition. But if one measures for example cohesion with variables of the parent class and methods of the child class, one is actually measuring how the parent and the child are coupled.

The following sets are useful: The set of locally defined instance variables

$$\mathfrak{F}_l(\mathfrak{c}) = \text{vars}(\mathfrak{c}) \backslash \mathfrak{F}_{\text{sta}}$$

the set of all instance variables

$$\mathfrak{F}_a(\mathfrak{c}) = \text{vars}(\text{parents}^*(\mathfrak{c})) \backslash \mathfrak{F}_{\text{sta}}$$

the set of inherited instance variables

$$\mathfrak{F}_i(\mathfrak{c}) = \mathfrak{F}_a(\mathfrak{c}) \backslash \mathfrak{F}_l(\mathfrak{c})$$

and the set of all instance variables ignoring the inherited private variables

$$\mathfrak{F}_{inp}(\mathfrak{c}) = \mathfrak{F}_l(\mathfrak{c}) \cup (\mathfrak{F}_i(\mathfrak{c}) \backslash \mathfrak{F}_{\text{priv}})$$

The locally defined private variables are not ignored in the last set. It may also be necessary to consider variables that are defined locally in the parent class.

2.3.6 Methods

When considering methods for classes, there are similar issues with inheritance as with instance variables. There is also the additional problem of "simple" or otherwise special methods. Simple methods are methods like getters and setters, which make the cohesion lower, because they only access one variable. On the other hand, almost all classes need this kind of methods.

One decision that should be made is whether or not constructors should be included in the set of methods. In the present study it is not done, because in Java it can be considered that constructors automatically use every instance variable.

The most obvious set for methods is the set

$$\mathfrak{M}_l(\mathfrak{c}) = \text{meths}(\mathfrak{c}) \backslash (\mathfrak{M}_{\text{abs}} \cup \mathfrak{M}_{\text{sta}})$$

which is the set of locally defined methods. But this set includes private and protected methods, which are not part of the interface, and as implementation details those should not be measured. The set of public methods is

$$\mathfrak{M}_p(\mathfrak{c}) = \mathfrak{M}_l(\mathfrak{c}) \backslash (\mathfrak{M}_{\text{priv}} \cup \mathfrak{M}_{\text{prot}})$$

If inheritance is taken into account, the set of all public methods is

$$\mathfrak{M}_a(\mathfrak{c}) = \text{resolve}(\mathfrak{S}, \mathfrak{c}) \backslash (\mathfrak{M}_{\text{prot}} \cup \mathfrak{M}_{\text{priv}})$$

and the set of inherited public methods is

$$\mathfrak{M}_i(\mathfrak{c}) = \mathfrak{M}_a(\mathfrak{c}) \backslash \mathfrak{M}_l(\mathfrak{c})$$

When a method is defined, there are three possibilities: It defines some new feature, implements some abstract method, or extends some existing method by redefining it and calling the old one with `super`-call. The two last cases might overlap. Notice also that $\mathfrak{M}_i(\mathfrak{c})$ is not the same as the set of all methods in the parent class $\cup_{\mathfrak{c}' \in \text{parents}(\mathfrak{c})} \mathfrak{M}_a(\mathfrak{c}')$, because if some method is redefined, the old version is not included in $\mathfrak{M}_i(\mathfrak{c})$.

While considering inheritance, it might be useful to have different sets with protected methods included, since these form a part of the interface towards inheriting classes.

The set of methods that use i instance variables is defined as

$$\mathfrak{M}_a(i, \mathfrak{c}) = \{\mathfrak{m} \in \mathfrak{M} |\ |\text{uses}^*(\mathfrak{c}, \mathfrak{m})| = i\}$$

or

$$\mathfrak{M}_p(i, \mathfrak{c}) = \{\mathfrak{m} \in \mathfrak{M} |\ |\text{uses}^*(\mathfrak{c}, \mathfrak{m}) \cap \mathfrak{M}_p(\mathfrak{c})| = i\}.$$

Clearly, the set of instance variables that a method uses, depends on the class that has gotten the method by inheritance. One use for this definition is to ignore methods that do not use any instance variables at all. We call methods that do not use any variables as *simple methods*. If a method has no functionality, it is called a *stub method*.

2.3.7 Calculation of cohesion

The calculation of cohesion from the cohesion graph should only depend on the structure of the graph. For a bipartite graph $G = (V_1 \cup V_2, E)$, where V_1 and V_2 are disjoint vertex sets, LCOM* is simply:

$$LCOM(G) = 1 - \frac{|E|}{|V_1||V_2|}$$

TCC is calculated as:

$$TCC(G) = 1 - \frac{|\{(a_1, a_2) \in V_2 \times V_2 \mid E(a_1) \cap E(a_2) \neq \emptyset\}| - |V_2|}{|V_2|^2 - |V_2|}$$

Here $E(a)$ denotes the set of edges in graph G connected to node a. This gives the ratio of method pairs sharing instance variables. The definition of TCC is symmetric so the places of methods and variables can be switched. This reversed calculation method is called rTCC.

$$rTCC(G) = 1 - \frac{|\{(a_1, a_2) \in V_1 \times V_1 \mid E(a_1) \cap E(a_2) \neq \emptyset\}| - |V_1|}{|V_1|^2 - |V_1|}$$

Calculation of CBMC (Cohesion Based on Member Connectivity) is more complicated. A *cut-set* of a graph is a set of nodes such that removing these nodes would make the graph disconnected. Then let $mcut(B, G)$ be the set of minimal cut-sets in graph G, where there is an additional restriction that only nodes from set B can be removed. Let also $conn(G)$ be the set of connected components in bipartite graph G. Then,

$$\begin{aligned}
CBMC(G) &= 1, && \text{if } |V_1||V_2| = |E| \\
CBMC(G) &= 0, && \text{if } |\,conn(G)| > 1 \\
CBMC(G) &= \max_{C \in mcut(V_1,G)} \sum_{G' \in conn(G \backslash C)} \frac{|C| CBMC(G')}{|V_1|\,|\,conn(G \backslash C)|}, && \text{otherwise.}
\end{aligned}$$

It is also useful to define a more coarse grained metric that measures the number of connected components in the cohesion graph. This metric is called the number of components, NOC.

For the instance variable usage of methods, no variations are considered and it is always computed using $(\mathfrak{m}, \mathfrak{f}) \in \text{Use}(\mathfrak{c}) \Leftrightarrow \mathfrak{f} \in \text{uses}^*(\mathfrak{m}, \mathfrak{c})$. Then the cohesion measure is denoted by $\text{CALC}_{\text{vars,meths}}$, where CALC is LCOM, TCC or CBMC, and the subscripts vars and meths use the same abbreviations as the measures that count methods or variables. The value 0.0 stands for the best cohesion for all metrics, and 1.0 indicates the worst cohesion. For example, $\text{LCOM}_{\text{l,p}}$ is LCOM with local variables and non-private methods and $\text{LCOM}_{\text{a,a}}$ is LCOM with inherited variables and non-private inherited methods.

2.3.8 Other metrics

Metrics related to the number of fields in a class are NV = $|\mathfrak{F}_n(\mathfrak{c})|$, NAV = $|\mathfrak{F}_i(\mathfrak{c})|$, NNPAV = $|\mathfrak{F}_{inp}(\mathfrak{c})|$ and NSTAV = $|\text{vars}(\mathfrak{c}) \cap \mathfrak{F}_s|$. The metrics related to the number of methods for a class $\mathfrak{c}$ are NNM = $|\mathfrak{M}_n(\mathfrak{c})|$, NNPM = $|\mathfrak{M}_{np}(\mathfrak{c})|$, NAM = $|\mathfrak{M}_i(\mathfrak{c})|$, NNSM = $|\mathfrak{M}_n(\mathfrak{c}) \backslash \mathfrak{M}_t(\mathfrak{c})|$, NNSNPM = $|\mathfrak{M}_{np}(\mathfrak{c}) \backslash \mathfrak{M}_t(\mathfrak{c})|$ and NNSNAM = $|\mathfrak{M}_i(\mathfrak{c}) \backslash \mathfrak{M}_t(\mathfrak{c})|$. Further NABSM = $|\text{meths}(\mathfrak{c}) \cap \mathfrak{M}_a|$ and NSTAM = $|\text{meths}(\mathfrak{c}) \cap \mathfrak{M}_s|$ are metrics measuring the number of abstract and static methods.

See Table 2.1 for the rest of abbreviations for the metrics used.

NV	Number of locally defined variables.
NAV	Number of all variables, including inherited ones.
NNPAV	Number of all variables, including inherited ones, but not private inherited ones.
NNM	Number of locally defined methods.
NNPM	Number of locally defined non-private methods.
NAM	Number of inherited and locally defined methods.
NNM w/ n	Number of locally defined non-private methods that access n instance variables.
NAM w/ n	Number of inherited and locally defined methods that access n instance variables.
NNSM	Number of locally defined methods ignoring simple methods.
NNSNPM	Number of locally defined non-private methods ignoring simple methods.
NNSNAM	Number of inherited and locally defined methods ignoring simple methods.
DIT	Depth in inheritance tree.
NSC	Number of subclasses.
NC	Number of constructors.
CC	McCabe's cyclomatic complexity.
ACC	Cyclomatic complexity per method in a class.
SZ	Size of a class, ie. number of statements in a class.
NSTAV	Number of static variables.
NSTAM	Number of static methods.
NABSM	Number of locally defined abstract methods.

Table 2.1: Abbreviations for various metrics.

2.4 Flattened LCOM

The *flattened* LCOM is calculated by taking inherited fields and methods into account. More precisely, it is the metric $\mathrm{LCOM}_{\mathrm{a,a}}$ for corresponding sets $\mathfrak{F}_a$ for all variables and $\mathfrak{M}_a$ for all methods. This metric corresponds to the run-time idea of class in the object oriented languages, where each object of the class includes all inherited fields, and each inherited method can be called from an object of the class.

It is natural to begin the analysis of internal cohesion with $\mathrm{LCOM}_{\mathrm{a,a}}$ because all components of internal cohesion are included there. The cohesion of a class can be decomposed into four parts: $\mathrm{LCOM}_{\mathrm{l,p}}$ measures the local component, $\mathrm{LCOM}_{\mathrm{i,i}}$ measures the cohesion of the unchanged component inherited from the parent, and $\mathrm{LCOM}_{\mathrm{i,p}}$ and $\mathrm{LCOM}_{\mathrm{l,i}}$ measure the connection between new and old parts of the class. $\mathrm{LCOM}_{\mathrm{a,a}}$ cannot directly

be calculated from these metrics, because the sizes of different components vary.

2.4.1 Interpretation of flattened LCOM

The only case where consideration of the inheritance is not really needed is the case where classes inherit directly from the base class `java.lang.Object`.

A problem with flattened LCOM is that it depends so much on the base class. For example, the class `java.lang.Object` includes just four methods that do not use any variables. This automatically decreases the cohesion of all classes, when inheritance is included. So, if the cohesion value is low, it is necessary to find out if it is caused by a base class, and by which base class. Because of this, it is useful to check the difference of the cohesion values between the base class and the child class.

The following findings should be established

- The results are difficult to interpret.
- The cohesion of the combined class is usually lower than the cohesion of parts.
- A child class usually has lower cohesion than the base class.
- Cohesion is highly dependent on the size of the classes.

To understand internal cohesion better, it is useful to inspect local cohesion, and the connections between the base class and its child classes.

2.4.2 Practical tests with flattened LCOM

A large number of classes (from Eclipse and JDK) was measured using the flattened LCOM metric. Classes with different metric values were inspected until it was difficult to find new phenomena associated with cohesion.

Maximal LCOM

Reasons for maximal $\mathrm{LCOM_{a,a}}$ are mostly the same as for maximal $\mathrm{LCOM_{l,p}}$. Several classes just define more instance variables without providing any local instance methods. A typical example of this is `ReverseMap.Entry` in unit `jdt.internal.ui.javaeditor.CompilationUnitDocumentProvider`. It has two instance variables. Otherwise it just extends `java.lang.Object`. These are the data classes.

High LCOM values (0.8-0.99)

The class `java.beans.BeansAppletStub` has 6 locally defined methods and 5 locally defined variables for this class. Four of the methods are accessors, and two do nothing. $LCOM_{a,a}$ is 0.92. This class implements an interface, and therefore some methods do not use any variables. The class is used in the implementation of another class, and is not a part of the public interface for the package. Because of this, the class cannot be measured independently of its client, so $LCOM_{a,a}$ does not apply to this class.

The class `org.eclipse.ui.views.framelist.FrameAction` includes in total 15 variables and 40 methods. $LCOM_{a,a}$ is 0.93. This class implements a completely orthogonal extension of its parent class. The parent class has a similar cohesion. Only two of the variables are locally defined. The class is abstract, and some of its methods will be overridden by the child classes. For this reason, its bad cohesion value does not mean that there is something wrong in the implementation.

The class `com.sun.java.swing.plaf.windows.XPStyle.GlyphButton` has 3 locally defined and 125 inherited variables, and 5 locally defined and 462 inherited methods. $LCOM_{a,a}$ is 0.97. The LCOM values are always high, no matter how they are measured. The class has 14 disjoint components when considering all variables and methods. Two components are caused by unused variables `vertical` and `glyphImage` (used only in the constructor), and others are inherited from `javax.swing.JButton`. This class is an example of bad cohesion. The bad cohesion is caused by the size of the inherited part of the class. This example shows why it does not make sense to apply LCOM to large classes: In order to be cohesive, the methods of this class would have to use dozens of fields.

The class `jdt.internal.ui.dnd.BasicSelectionTransferDragAdapter` of Eclipse has one locally defined variable and 4 locally defined methods. Only one of the methods uses the variable. $LCOM_{a,a}$ is 0.88, but if the simple methods are ignored, it becomes 0.0 ($LCOM_{a,ans}$). The three simple methods are used via virtuality. The $LCOM_{a,ans}$ value seems to be a descriptive measure for the cohesion of this class.

Another Eclipse class `core.internal.content.LowLevelIOException` has one locally defined variable, and one accessor method. This class has $LCOM_{a,a}$ 0.8. The parent class `java.lang.Throwable` has $LCOM_{a,a}$ 0.77. This is an example of how an orthogonal extension can make the $LCOM_{a,a}$ higher. There is nothing wrong in the class itself. The class has low cohesion value because low-level data are attached to the class via inheritance.

Eclipse class `team.ui.synchronize.SubscriberParticipant` has 3 locally defined, and 7 inherited variables, and 18 locally defined and 18 inherited methods. $LCOM_{a,a}$ is 0.83. This class seems to be quite normal. Some of the inherited variables are used, and also some inherited methods use the

locally defined variables. $LCOM_{l,p}$ is 0.53. The relatively low cohesion is caused by the fact that the inherited variables are not used very much by the locally defined methods, and on the other hand, the inherited methods do not access the local variables that much. Also 22 of the methods use one or zero variables. The class has only one component, so it seems to be cohesive.

The class `com.sun.jmx.snmp.tasks.ThreadService` has 11 locally defined variables and 5 locally defined methods. $LCOM_{l,p}$ is 0.64 and $LCOM_{a,a}$ is 0.80. This class inherits directly from `java.lang.Object`. If simple methods are ignored, the inherited methods from `java.lang.Object` are removed, and the values are the same. Some of the instance variables are used only by an inner class, so the cohesion cannot be properly measured with $LCOM_{a,a}$.

Average LCOM (0.5-0.8)

Eclipse class `ui.internal.components.framework.SingletonFactory` has 2 locally defined variables as well as 2 locally defined methods. $LCOM_{a,a}$ is 0.75 and $LCOM_{l,p}$ is 0.25. This shows that the inherited simple methods from `java.lang.Object` can make a big difference in the measure, at least when the class is small. Clearly, the actual cohesion of the class is good.

The class `com.sun.jmx.snmp.IPAcl.AclImpl` has one inherited and 2 locally defined variables, and 11 locally defined and 6 inherited methods. $LCOM_{a,a}$ is 0.65. The inherited methods do not use the local variables, but the new methods use the inherited variables. When simple methods are ignored, the value is 0.57. The class has only one component, so it seems to have good cohesion.

The class `org.eclipse.jdt.core.dom.DocCommentParser` has a total of 42 variables and 3 locally defined methods. Two of the methods use most of the variables. The `toString`-method uses some of the variables. $LCOM_{a,a}$ is 0.65 and, if the simple methods are ignored, it is 0.30. The class has good cohesion.

In the class `org.apache.tools.ant.taskdefs.Exit.NestedCondition`, there are 2 inherited variables and 26 methods out of which one is defined locally. $LCOM_{a,a}$ is 0.58, almost the same as the value for the parent class. Most of the inherited methods use one variable, and because the class does not have any new variables, but just adds a method that uses one variable, and therefore the cohesion does not change. The class has two components, so it does not seem to have good cohesion.

Comparison between cohesion of base classes and their child classes

There are 16620 classes in the sample for which $LCOM_{a,a}$ is applicable (that is, there are more than zero instance variables), if the simple methods are ignored. The local version is applicable in much fewer cases, so it is useful to study what kind of results are given for the classes where the local case is not applicable.

In 4138 cases there are no local variables. Out of these, in 2892 cases, cohesion is the same or almost the same as the parent's cohesion. In 638 cases, the $LCOM_{a,a}$ is higher than the value for the parent and in 608 cases the $LCOM_{a,a}$ is lower.

In the cases where the cohesion is similar in the parent and child, the child class might override a method, and then this new method uses the same variables. Also often there is only one variable, and this is used by every method in which case the cohesion remains.

The class `javax.sound.midi.Track.ImmutableEndOfTrack` is an example of a class, where the cohesion is lower than for its parent. This class overrides a method with one that uses no variables. The usual reason for the cohesion getting lower in the child class is just having methods that do not use many of the inherited variables.

A typical class that has a better cohesion than its parent is `ValueIterator` in unit `java.util.HashMap`. It implements an abstract method that uses all inherited instance variables. The implementation of the abstract method is done by using a protected method of the parent class.

In 2717 cases there are no local non-simple methods. Of these, in 138 cases the cohesion of the child is better, in 395 cases worse than in the parent, and in 2184 cases there is no difference. In this last category, there are usually no local variables. It is also possible that for example the parent had extremal cohesion 0.0 or 1.0.

A typical case, where the child has a worse cohesion, is `EventException` in package `org.w3c.dom.events`, where there is just one public variable added. This is an example of a base class that has data that is not directly related to the data that child classes have.

An example of a class that has better cohesion than its parent is Eclipse class `ui.internal.WindowSelectionService`. It inherits from an abstract class `AbstractSelectionService`, and implements a protected method to access its instance variable.

Discussion

At least in principle, $LCOM_{a,a}$ seems to be the best LCOM variant to represent the cohesion of a class, because the inheritance is included in it, and therefore the whole object is considered. Ignoring the simple methods seems

	0-0.1	-0.2	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0	1.0
0.0-0.1	3648	78	139	169	252	554	1051	1513	1883	404	703
0.1-0.2	3	22	2	3	3	0	2	3	4	1	0
0.2-0.3	0	5	9	12	5	6	4	11	2	1	0
0.3-0.4	1	1	3	26	5	36	14	10	12	0	0
0.4-0.5	0	0	1	25	56	36	19	16	9	2	0
0.5-0.6	2	8	15	17	31	210	155	92	67	0	1
0.6-0.7	0	0	3	2	7	41	395	278	84	8	0
0.7-0.8	0	0	0	0	1	13	116	1535	668	32	4
0.8-0.9	1	0	0	0	7	9	71	354	2358	506	5
0.9-1.0	0	0	0	0	0	5	33	52	318	2541	2
1.0	7	2	1	1	4	4	9	22	46	17	113

Table 2.2: Distribution of the $LCOM_{a,a}$ values for classes and their parents. The cohesion of the 16620 sample classes have been evaluated. Rows stand for the $LCOM_{a,a}$ value ranges of the parent classes and columns for the $LCOM_{a,a}$ value ranges of the corresponding child classes.

a particularly good idea here, since the base classes always have such methods, in particular the class `java.lang.Object`.

One problem with the flattened LCOM is that the result of the measure usually depends strongly on the parent class. A solution is to calculate the difference between cohesion of the parent class and the child class. Then the highest differences would be for classes that inherit `java.lang.Object`, and the measure would be similar to $LCOM_{l,p}$. This is caused by the fact that the classes being deeper in the inheritance hierarchy have usually a high lack of cohesion so the difference cannot be very high there.

Table 2.2 demonstrates the correlation between the flattened LCOM value for parent and child classes. The table shows that the values of $LCOM_{a,a}$ are most of the time similar to values of parents, except for classes where the parent has $LCOM_{a,a}$ value of 0.0, which usually means that the parent is `java.lang.Object`.

Another problem is that the value $LCOM_{a,a}$ depends so much on the size of the class, especially the number of instance variables. This is the same problem as with $LCOM_{l,p}$, but it is even worse, because including the inherited part makes the classes bigger. Thus, there is a need to develop measures that do not depend on the size of classes so much.

There are two possible viewpoints for a class. The first is that it is a kind of an independent component that can be measured without considering the inheritance. This is the viewpoint of $LCOM_{l,p}$. The second is that the inheriting class contains the base class, and all the features of the base class

should be used when measuring the cohesion of the child. The metric suited for this viewpoint is $LCOM_{a,a}$.

The metrics $LCOM_{i,p}$ and $LCOM_{l,i}$ measure what kind of inheritance relation the class uses. These variations are more natural than the variations with all methods with normal variables, and all variables with normal methods. The two variations also show more clearly the differences between different classes. If these metrics indicate that the relation is tight, $LCOM_{a,a}$ should give a better idea on the cohesion of the class.

2.5 Local LCOM

The traditional way to define LCOM is to consider only the locally defined variables and methods. We call this the local LCOM. In this section, we try to find out what are the weaknesses and advantages of local LCOM. The test set consists of the JDK and Eclipse source codes.

The most natural way to define local LCOM is $LCOM_{l,p}$. Other possibilities are the versions without indirect computation, and the version where also private methods are included in the computation. First we explain why $LCOM_{l,p}$ can be considered the most natural definition of local LCOM. Then it is studied for what kind of classes the metric is not applicable at all. For the applicable classes, it is evaluated whether the results given by $LCOM_{l,p}$ are reasonable.

2.5.1 Indirect and direct usage of instance variables by methods

The reasons why indirect calculation should be preferred is that when observing a class from outside, the method can be thought to include the functionality of the called internal methods. Of course now the problem is that if fields are internal, why should they be measured.

In practice indirect computation is better, because it gives better LCOM values. Also if accessors of methods are used internally instead of instance variables, direct computation does not make any sense.

2.5.2 Dynamic type

Another point of variation is handling the dynamic type. Each object actually has a constant that represents its class at the run time. The use of this implicit dynamic type variable can be considered as an additional instance variable in the measurements. Basically there are two possibilities: this-variable is used with instanceof-expression, or a virtual method is called using this-variable. A problem here is that in Java, all methods are virtual in principle, but most are not overloaded.

2.5.3 Private methods and variables

When comparing all methods ($LCOM_{l,l}$) and non-private methods ($LCOM_{l,p}$), the two measures give the same value in 15782 of all 21036 classes of the sample set. In most of these cases, there are no private methods, so the values are the same because of that reason. In 1364 cases cohesion values are higher if all methods are ignored, and in 2826 cases higher values are got by ignoring the private methods. The classes, where the values are equal, are quite small, and the classes, where values differ, are bigger. There are no significant differences between classes where $LCOM_{l,l}$ is higher and where it is lower than $LCOM_{l,p}$.

The alternative of ignoring the private methods is more natural, because the non-private methods define the interface and the private methods are just implementation details. Because the indirect variable usage calculation is used, the private methods can be safely ignored. If the indirect calculation was not used, the situation would be different. Then it would probably be best to include the private methods into the calculations.

There are some cases where the private methods are used by something that is not defined in the class itself, for example the serialization methods like `readObject`. Another possibility is using reflection to call the private methods. It might be useful to include methods like `readObject` or `finalize` into the computation. Adding support for reflection does not seem to be a good idea.

Protected methods can also be seen as a problem, because they should be used with inheritance, and the local metric does not consider inheritance. We choose not to include the protected methods, because this makes more sense in Section 2.4, when inheritance was considered.

The situation with private variables is similar to that of private methods in the sense that both variations give similar results. But now, including private variables into the computation of cohesion is more natural, since all variables should be defined private anyway.

2.5.4 Trivial classes

There are two cases where $LCOM_{l,p}$ is inapplicable for a class. The first is the case when a class has no locally defined variables, and the second is that a class has no locally defined methods. In both cases, it is said that the class is *trivial*.

For 7771 (37%) classes out of 21036, there are no such variables, and for 4208 (20%) classes there are no locally defined public methods. In total, there are 8904 (42%) classes without either locally defined variables or locally defined methods or both. Thus, $LCOM_{l,p}$ is only applicable to 63 percent of all cases, and $LCOM_{a,a}$ is applicable to 80 percent of all cases.

In 4208 (20%) classes there are no instance methods. Most of these classes (3076) have no instance variables either. These classes include static methods, static variables, and/or inner classes. One scenario also is that a set of classes is defined, and then their dynamic type is used as a tag meaning that the type information is used as an instance variable value. This is the case e.g. with the exception classes.

Over a third of the classes (7771) have no locally defined instance variables at all. To give an idea, what kind of classes these are, one can inspect the suffixes of the classes. The suffixes for all measured classes are: 1148 Action, 558 Provider, 556 Exception, 583 Impl, 435 Page, 404 Handler, 317 Listener, 311 Dialog, 285 Info, 282 Factory, 279 I, 237 Adapter, 232 Event, 285 Info, 288 Manager, 249 Helper, and 211 Messages. In 690 classes there is a unique suffix. There are 1582 different suffixes. Of the classes, that have no locally defined instance variables, 491 have suffix Action, 401 Exception, 307 Provider, 253 Handler, 222 Helper, 210 Messages, 207 Listener, 186 Factory, 132 Util, and 111 Adapter. From these suffixes one can confirm that the reasons for these classes are that the class either implements a functional pattern (for example Action-classes), or it is used as dynamic type information (e.g. Exception-classes).

The 4695 (22 %) classes having methods but no variables are classes that are used like functions. Similarly, classes having variables but no methods are used as pure data.

2.5.5 Normal classes

In $LCOM_{l,p}$ only locally defined variables and methods are used for calculating the cohesion. For the usage selection, we apply indirect computation, which also uses inheritance. When analyzing the indirect computation it is necessary to inspect the inherited methods, because these methods might call virtual methods from the local class. In turn, these methods then use locally defined variables.

From Tables 2.3, 2.6, 2.8 and 2.9 it can be seen that $LCOM_{l,p}$ correlates best with numbers of locally defined variables (V) and methods (M), and with the value of $LCOM_{a,a}$. This seems quite logical, because $LCOM_{l,p}$ is in a way included in $LCOM_{a,a}$, and only locally defined variables and methods are used in the calculation. $LCOM_{l,p}$ does not correlate much with numbers of inherited variables (IV) or methods (IM), and also not with $LCOM_{i,p}$ or $LCOM_{l,i}$.

Maximal local LCOM

The first question is what kind of classes have the maximum lack of cohesion (i.e. LCOM = 1). There are 187 (less than 1%) such classes. It is odd that

Range	IV	V	IM	M	Classes
Trivial					8903
1.0	9.6	2.3	31.1	3.1	187
0.9-1.0	14.7	13.8	61.8	31.2	323
0.8-0.9	9.6	8.9	38.7	15.7	1152
0.7-0.8	7.3	6.8	28.5	11.9	1250
0.6-0.7	6.3	4.9	25.1	8.7	1617
0.5-0.6	5.8	3.5	22.4	6.1	1882
0.4-0.5	4.5	4.3	19.4	8.2	740
0.3-0.4	5.5	2.8	21.4	5.4	1115
0.2-0.3	4.5	2.7	19.8	5.6	543
0.1-0.2	5.5	2.7	20.6	6.0	451
0.0-0.1	5.1	1.7	20.0	3.3	2873
Cor.	0.10	0.39	0.09	0.34	

Table 2.3: Distribution of classes in JDK and Eclipse based on their $LCOM_{l,p}$ values. The table also shows average numbers of variables and methods. Rows show the average number of inherited variables (IV), local variables (V), inherited methods (IM), local methods (M) and number of classes for a specific range of $LCOM_{l,p}$. The bottom row shows the correlation with the attribute and $LCOM_{l,p}$.

a class has instance variables, and then it has instance methods which do not use these instance variables. In the following, we analyze 5 such cases. Most of these classes are found in Eclipse.

Eclipse class `pde.internal.ui.model.plugin.PluginDocumentHandler` has protected methods that access the only instance variable declared, and one public method that uses only inherited variables. In our opinion, this class is well designed and the reason for the bad LCOM value is that the class is designed to be used via inheritance.

Class `java.util.AbstractList` is similar. It has one instance variable that is used only in an inner class (an iterator) and the idea seems to be that an inheriting class can control the operation of the inner class via that variable. One can also say that in this case the class is designed to support its inner classes.

Another example is `org.eclipse.jdt.core.dom.Name`. It has one variable, `index`, that is visible to the package, and then there are methods that either use the inherited variables or `this`-variable to test the type of the object. In our opinion, this class is not well designed – instead of exposing the instance variable `index` to the package, one should provide methods to manipulate that variable property and thus fulfill the usual requirement of encapsulation in object oriented design.

Class `org.eclipse.compare.CompareViewerPane` has an instance variable but for using the instance variable, the class provides static methods. It is hard to find a good reason for this.

The Java RMI has class `UnicastRemoteObject`, which plays a central role in RMI. It declares 3 instance variables but none of its public instance methods uses them. The actual use of the instance variables is related to serialization and reflection. Since this a rather low-level system class, it is difficult for us to say whether the bad $\mathrm{LCOM}_{\mathrm{l,p}}$ value in this case is a sign of bad design.

As a summary, the problematic cases can be detected using the following heuristics:

- Check if a class has non-private fields.
- Check if a class has both static and normal members.

If a class has a public field, it would be inconsistent to also have private fields. If a class has a public field, it would be inconsistent to have accessors for it. In the test material, there were 16 classes with both public and non-public fields. There were 32 classes with only public fields, 527 classes with only private fields, and 193 classes with no fields at all. Having both private and public fields was detected to be a design anomaly.

Sometimes in larger classes, it might be logical to have both normal and static members. The most illogical case would be that there are normal fields

and only static methods. When checking for normal and static members, no design anomaly was found. Apparently not much effort is given to keep static and normal members separate. On the other hand, there was no such case where there would be normal fields but only static methods.

High local LCOM

The above cases are quite pathological, and the problems that cause them can possibly be detected in some other ways, too. Next we study what kind of classes get very high lack of cohesion, but not the maximal one. The $LCOM_{l,p}$ values for all the classes studied next are above 0.9. The first observation, based on Table 2.3, is that these classes tend to be quite big (the average numbers of instance variables and methods are 13.8 and 31.2, respectively).

In package called `com.sun.org.apache.xml.internal.serializer`, in class `ToXMLSAXHandler`, there are one variable and 37 methods. Its $LCOM_{l,p}$ is 0.95, which is due to the fact that most of the methods use its parent class services to implement the required functionality. In this sense, this class is badly implemented, since most of the functionality of this class could be pushed up in the inheritance hierarchy.

This kind of suspicious methods can be identified as follows:

- They do not use any locally defined instance variables.
- They are not called via dynamic binding.
- Even if they were called using dynamic binding, if they use the instance variables of the parent they are suspected.

When checking for methods that use parent fields directly, 821 such methods were found. There were 11475 out of 13430 methods that used local fields internally. It was found out that 341 methods used only the parent directly, which is a *design tendency*, the opposite of design anomaly. Dynamic overloading and checking if the class has any local fields could not generate an anomaly. The result was that supposedly problematic cases were found, but no design anomaly.

The class `javax.swing.JTable.AccessibleJTable` has 6 variables and 53 methods. Of these, 32 methods do not access any variables, and 11 access only one variable. This class is an inner class, and it uses heavily the attributes of the outer class. The class is designed to use its outer class – making it hard to analyze the class with this metric.

The class `org.eclipse.jdi.internal.VirtualMachineImpl` has 46 variables and 75 methods. Its $LCOM_{i,p}$ is 0.54, so it uses the features of its parent. Its $LCOM_{a,a}$ is 0.88. To see, why the cohesion is low, we can check how many connected components the cohesion graph of the class has. If we

ignore methods that do not use any variables from the cohesion graph, then there are still 15 disjoint components in the class. There are 8 components that implement singleton patterns. Some components are for variables that have only getters and setters. The huge number of components suggests that this class (or its parent class) is badly designed, but its rather good $\mathrm{LCOM}_{\mathrm{i,p}}$ suggests that it is designed to strongly lean on its parent class.

The class `org.eclipse.ui.internal.WorkbenchWindow` has 37 variables and 70 methods. Of the methods, 37 methods use exactly one variable. All cohesion measures give bad values for this class, except that if the simple methods are ignored, then there is only one component in the class. The class `javax.swing.text.JTexTComponent` is similar. It has 26 instance variables and 67 own public methods. Most of them use only a few variables and the class consists of several components. The class `org.eclipse.jface.text.TextViewer` has 55 variables and 91 methods of which many are getters and setters. Over 50 methods use only one instance variable. In our opinion, the main reason for bad LCOM values in the above cases is simply the large fraction of simple methods and accessors in these classes. It is not reasonable to expect all methods to operate with majority of the instance variables.

Eclipse class `ui.texteditor.templates.TemplatePreferencePage` has 12 variables and 8 methods. There are three disjoint components in the class. One private function is called only from an anonymous inner class and this is not measured. This class is partly designed to support its inner classes, but most of the methods are designed to strongly use the parent class.

Eclipse class `jdt.internal.codeassist.InternalCompletionProposal` has only one public setter method, the others are protected. The methods are mostly getters and setters. There are 11 variables. This class is designed to be used via inheritance. Some protected methods are actually used by other classes in the package, not by the inheriting classes, which is somewhat confusing.

Eclipse class `pde.internal.ui.model.plugin.PluginDocumentNode` has 10 variables and 29 methods. The methods are mostly getters and setters. In fact, all methods use exactly one variable. This means that the class has 10 disjoint components. The data might still have something in common, but it cannot be seen from this class, rather from classes that are using this class. This class is an example of *data class*.

The class `org.eclipse.swt.internal.ole.win32.COMObject` has 82 methods and one variable. Of the methods, 80 are stubs that do not use any variables (those return a constant value). Otherwise the class has perfect cohesion. For this kind of classes, ignoring simple methods makes a big difference. This class is designed to be used via inheritance (or more precisely, for mapping native objects into Java).

Average local LCOM

Eclipse class `jdt.debug.ui.launchConfigurations.AppletMainTab` has 11 variables and 10 methods. $\text{LCOM}_{\text{l,p}}$ is 0.8. The class uses many private methods. Most methods use only a few variables, and as is common with classes that have several variables, the LCOM value becomes high. If simple methods are ignored, $\text{LCOM}_{\text{l,p}}$ is 0.6. The class has two different components. This split comes from the base class. One can not conclude that this class is badly designed, although the number of variables is rather high – it is quite ordinary that all methods do not use all the variables.

Another Eclipse class `jdt.internal.debug.ui.JDIModelPresentation` has $\text{LCOM}_{\text{l,p}}$ 0.84. There are 4 variables and 8 methods. The value is high because 5 methods do not use any instance variables. Without these methods, the result would be 0.58. The class has numerous protected and private methods. $\text{LCOM}_{\text{a,ans}}$ is 0.67. It cannot be concluded that this class is badly designed.

The class `java.nio.ByteBuffer` has $\text{LCOM}_{\text{l,p}}$ value 0.86, and $\text{LCOM}_{\text{a,a}}$ value of 0.80. There are four different components in the class. The class is an abstract class, so it cannot be concluded that the class is badly designed. On the other hand, the classes that inherit from it, have similar characteristics. Some methods use instance variables, not via the self-object but via another object of type `ByteBuffer`. Perhaps the calculation should be refined so that all instance variables used from the class are counted, and not just the instance variables used from the object. The idiom for using private variables to implement this kind of methods seems to be quite common, because in Java it is possible to access private instance variables of any object of the class, if the method is defined in that class.

One special case is formed by the classes that have only accessor methods. These classes get LCOM value $1-\frac{1}{n}$, where n is the number of locally defined variables. For example if there are two variables, the cohesion is 0.5. The reason why this kind of classes are needed is that they collect two or more logically connected classes into one data-structure. The problem with them is that they do not have any operations working on the data, so they are helper data structures for grouping the data, and not really objects.

Eclipse class `jface.text.MarkSelection` has 3 variables and 4 methods. $\text{LCOM}_{\text{l,p}}$ is 0.67 and $\text{LCOM}_{\text{a,ans}}$ is 0.67. All the methods are getters. This is a case where TCC gives worse values. This class is not really cohesive as there are three separate components. In class `ContextService` in package `org.eclipse.ui.internal.contexts`, there are 2 variables and 14 methods. $\text{LCOM}_{\text{l,p}}$ and $\text{LCOM}_{\text{a,ans}}$ are 0.5, and $\text{TCC}_{\text{a,a}}$ is 0.36. Each method uses one variable, so the class has two components. These classes are not internally cohesive, but they might group the data in a way that is useful to the clients.

The class `javax.swing.text.ChangedCharSetException` has 2 variables and 2 getter methods. $\mathrm{LCOM}_{\mathrm{l,p}}$ is 0.5 and $\mathrm{LCOM}_{\mathrm{a,ans}}$ is 0.79. This is because the superclasses for exceptions have low cohesion.

In our opinion, most classes in the range 0.1-0.7 have no special problems and the problematic cases can perhaps be best found by calculating the number of components or by some other metric like TCC. The classes in this range have usually a small number of variables, so the LCOM values fall into this range even if most of the methods are just accessors.

Minimal local LCOM

The classes with maximal or almost maximal cohesion are almost always very small, but there are some exceptions. These classes are usually not very interesting, but there are lots of them so we study what they are like.

Of the 2873 classes (14% of all) with minimal local LCOM, many have only one variable. Complexity of the methods seems to vary. It cannot be concluded that all these classes are well designed, because they are perhaps too small to implement their own independent functionality.

Eclipse class `core.internal.resources.Marker` has 2 variables and 20 methods. Of the methods of this class, 17 use both variables. Another similar class is `internal.texteditor.quickdiff.ReferenceSelectionAction` in Eclipse. It has 3 variables and 2 methods. The methods are relatively complex, and there are two private helper methods.

The class `org.eclipse.jdt.ui.search.ElementQuerySpecification` has only one variable and one method. The classes that have maximal cohesion are usually small, as is obvious from the definition.

There are very few big classes that have maximal or almost maximal cohesion. One such class is `javax.swing.plaf.synth.ImagePainter`. It has 7 variables and 118 methods. Of the methods, 113 access every variable.

2.5.6 Discussion

The evaluation of the classes in the sample set indicates that a good $\mathrm{LCOM}_{\mathrm{l,p}}$ value for a class indicates that the class is well designed with respect to cohesion, but a bad value does not always indicate bad design. A very good $\mathrm{LCOM}_{\mathrm{l,p}}$ value typically indicates that the class has only a few instance variables. Often in inheritance, the extension is orthogonal or almost orthogonal, meaning that new features added to the inherited class do not lean on the features defined in the parent class.

It is characteristic for LCOM that bigger classes get low and smaller classes get high cohesion values. Comparing $\mathrm{LCOM}_{\mathrm{l,p}}$ with the inheritance based measure $\mathrm{LCOM}_{\mathrm{a,a}}$ tells that $\mathrm{LCOM}_{\mathrm{l,p}}$ gives better values when the extension is orthogonal, and similar values, if the inheritance is used a lot.

Also, it is quite rare that $\mathrm{LCOM}_{\mathrm{l,p}}$ gives worse values than $\mathrm{LCOM}_{\mathrm{a,ans}}$ for other reasons than as a result of ignoring simple methods. Ignoring the simple methods is one option when measuring cohesion. If the number of these methods is interesting, it can be measured by another metric.

For quite many classes the $\mathrm{LCOM}_{\mathrm{l,p}}$ value is bad, although there is only one component in the sense of $\mathrm{LCOM}_{\mathrm{l,p}}$. This is because such a class has rather many instance variables and the methods typically use only few of them. We find that in most cases this is not a sign of bad design.

Some of the common reasons for bad $\mathrm{LCOM}_{\mathrm{l,p}}$ values are:

- The class is designed to be used via inheritance.
- The class mainly uses inherited features to implement its own features.
- The class is designed to support inner classes.
- The class is an inner class designed to use outer classes.

Calculating the number of components might give a better idea about problems or lack of cohesion for big classes.

There were no cases where local LCOM could have found a design problem that could not have been found using more simple metrics such as size.

The key for understanding very low cohesion is inspecting methods that use only one or zero fields. If for some field there are many methods that only use that field, either the class is a *wrapper* for the field, or otherwise the methods are in the wrong place. Another situation for inconsistency is that a class has a lot of accessors, and a lot of other functionality. It does not seem likely that this kind of classes could be eliminated. Realising this can lead to better understanding how the object-oriented principle of encapsulated data should be used:

- It is not useful for basic data. Something like relational paradigm is better.
- Memoized data: this kind of optimization is sometimes useful, and can be hidden and handled using encapsulation. Perhaps there should be an own construct for safe memoized data.
- Data structures and associated algorithms: In these cases, the implementation and specification are quite far away from each other.

Because the principle of encapsulation is the most important motivation for the definition of internal cohesion, the above observations could explain why it is hard to find design problems using internal cohesion.

In the previous section, we found out that restricting to a local view of cohesion is not always a good idea. $\mathrm{LCOM}_{\mathrm{l,p}}$ did not work for cases where

inheritance was used a lot. In fact, the local component of the class is not an independent entity at all, it just tells how the class differs from the parent class.

2.6 LCOM and inheritance

In the following sections, the metrics $\text{LCOM}_{\text{i,p}}$ and $\text{LCOM}_{\text{l,i}}$ are considered. These two metrics are not really cohesion metrics, they measure the connectedness of a parent and its child class. They operate in a similar way as coupling metrics [16], but strong connectedness should not be considered harmful for them.

Table 2.4 shows that the distributions of the cohesion metrics vary quite a lot depending on the used variation. The number of trivial cases, where there are no instance variables or no methods that are related to the metric, is quite large for each metric variation. In the following, the reasons behind the observed cohesion values will be classified.

2.6.1 Sizes of parts

The edges of the cohesion graph can be divided into four parts:

- Edges internal to parent class.
- Edges internal to child class.
- Edges from methods in parent using instance variables of the child (using dynamic binding).
- Edges from methods in child using instance variables of the parent.

Now we need to answer with how large ratio does each of the four parts affect the final measure. Table 2.5 shows the relative sizes of different parts of cohesion graph when calculating $\text{LCOM}_{\text{a,ans}}$. The local part is the largest for high cohesion. When cohesion is low, the inherited part is the largest. Because usually the class inherits from `java.lang.Object` or has a large inherited part, the parts related to $\text{LCOM}_{\text{i,p}}$ and $\text{LCOM}_{\text{l,i}}$ are small on average.

2.6.2 Connection of parent and child classes

The metric $\text{LCOM}_{\text{i,p}}$ measures how a class extends its parent. When $\text{LCOM}_{\text{i,p}}$ is 1.0, the class defines an orthogonal extension. For high $\text{LCOM}_{\text{i,p}}$ values, the class is mostly orthogonal, and just relies on some part of the parent class. When $\text{LCOM}_{\text{i,p}}$ is low, features of the parent are extended by using the old ones, which basically is the idea of implementation inheritance.

Range	$LCOM_{l,p}$	$LCOM_{i,p}$	$LCOM_{l,i}$	$LCOM_{a,a}$	$LCOM_{a,ans}$
Trivial	8903	12250	7771	3592	4437
0.0 - 0.1	2873	430	42	69	2935
0.1 - 0.2	451	84	6	116	313
0.2 - 0.3	543	95	11	173	453
0.3 - 0.4	1115	182	48	255	939
0.4 - 0.5	740	131	43	370	889
0.5 - 0.6	1882	528	64	912	1753
0.6 - 0.7	1617	728	189	1860	2416
0.7 - 0.8	1250	820	283	3880	2353
0.8 - 0.9	1152	1331	940	5448	2587
0.9 - 1.0	323	1801	2231	3512	1952
1.0	187	2656	9408	849	9

Table 2.4: Distribution LCOM values for the JDK and Eclipse classes. Trivial classes are the ones without methods or instance variables. The sample set has a total of 21036 classes.

Range	$LCOM_{i,i}$	$LCOM_{i,p}$	$LCOM_{l,i}$	$LCOM_{l,p}$
0-0.1	19	6	3	72
0.1-0.2	11	6	7	76
0.2-0.3	18	6	6	70
0.3-0.4	34	7	5	54
0.4-0.5	34	8	6	52
0.5-0.6	28	10	8	54
0.6-0.7	48	8	8	36
0.7-0.8	48	11	12	29
0.8-0.9	56	11	12	21
0.9-1.0	77	6	10	7
All	43	8	8	41

Table 2.5: Proportion in percentages of different LCOM values from $LCOM_{a,ans}$. Results are given for different ranges of $LCOM_{a,ans}$.

Range	IV	V	IM	M	$LCOM_{a,a}$	$LCOM_{l,p}$	$LCOM_{l,i}$
1.0	8.4	1.8	34.2	3.6	0.83	0.22	0.59
0.9-1.0	22.5	5.0	79.9	9.8	0.90	0.49	0.80
0.8-0.9	13.4	3.6	48.0	8.0	0.86	0.41	0.74
0.7-0.8	12.5	3.6	45.9	7.5	0.81	0.37	0.74
0.6-0.7	8.6	2.1	38.9	6.6	0.76	0.27	0.58
0.5-0.6	6.2	2.0	22.1	4.9	0.73	0.21	0.56
0.0-0.5	4.9	1.6	15.5	7.8	0.62	0.14	0.49
Cor.	0.16	0.10	0.14	-0.03	0.54	0.17	0.01

Table 2.6: Average values of selected attributes for different $LCOM_{i,p}$ ranges, and the correlations of these attributes with $LCOM_{i,p}$.

As can be seen from Table 2.4, most of the $LCOM_{a,p}$ values are high, indicating rather low usage of inherited features. From Table 2.6 it is seen that $LCOM_{i,p}$ correlates best with $LCOM_{a,a}$. This is partially because $LCOM_{i,p}$ is a component when calculating $LCOM_{a,a}$. Correlation of $LCOM_{i,p}$ with other measures is quite small. In particular it is a surprise that there is no correlation with the number of locally defined methods. This is a big difference to the metric $LCOM_{l,p}$. A possible explanation for this is that the correlation between the number of methods and $LCOM_{l,p}$ is caused by the correlation of the number of methods and variables, and not because the number of methods would affect cohesion by itself.

No connection

There are 2656 classes in the sample set which do not use inherited variables (not even indirectly via calling inherited methods), but instead make completely orthogonal extensions. Listing the most common suffixes gives some ideas of their nature: 444 has suffix Action, 192 Provider, 172 Impl, and 110 Exception.

The exception classes inherit from the class `java.lang.Throwable`, which has four instance variables. Two of them are related to stack traces, one is for linking exception causes, and one is for a detailed error message. The inherited detailed message variable is rarely used. The methods are mostly just accessors for locally defined variables.

The action classes are mostly from Eclipse, and they inherit from the class `org.eclipse.jface.Action`. This class has several instance variables that can be used to identify or describe the action, which is a kind of callback with a `run`-method. The inheriting classes usually implement this method, which naturally does not use the instance variables defined in the base class.

The class `org.eclipse.ui.dialogs.FilteredList` is an example of a

larger class that makes an orthogonal extension. It has three disjoint components, while its parent `org.eclipse.swt.widgets.Composite` has only one. The parent class is needed in creation of child widgets, but this is done in the constructor. The `Composite`-nature is used by the child widget, but not directly by the methods.

A typical example of completely orthogonal extension is a class, where the parent includes some information that is not directly related to the core functionality of the child class, for example the name for the action in the `Action` classes.

Loose connectedness (0.5-0.99)

The class `org.eclipse.jdt.core.dom.BooleanLiteral` has 8 inherited variables and 10 local methods. $LCOM_{i,p}$ is 0.93. Of the methods, 6 use no variables at all. $LCOM_{l,p}$ is 0.6 and $LCOM_{i,i}$ is 0.88. Some inherited variables are used by a copy method, and others are used by a mechanism that signals changes. The inherited part is used to help implementing observer pattern and it is needed for defining a generic method for copying.

The class `org.eclipse.ui.views.navigator.SortViewAction` has 15 inherited variables and one local method. This method uses one of the inherited variables and the only locally defined variable, so $LCOM_{i,p}$ is 0.93 and $LCOM_{l,p}$ is 0.0. This is an action class, which implements a function. The parent class is used to define an instance variable, that is used as an argument for the function. The base class is used to support implementation of child classes. This is an example of a class implementing a minor variation of a rather large base class.

In package `org.eclipse.jdt.internal.corext.refactoring.code`, in class `AstMatchingNodeFinder.Visitor` there is one inherited variable and 84 locally defined methods. $LCOM_{i,p}$ is 0.99. The class is almost a pure orthogonal extension. $LCOM_{l,p}$ is 0.01. In this case, the cohesion of parent is very low, since only one of its methods uses the defined variable. This is an example of a rarely used variable.

The class `javax.swing.text.AbstractDocument.LeafElement` has 2 inherited variables and 10 local methods. $LCOM_{i,p}$ is 0.9 and $LCOM_{l,p}$ is 0.8. Technical methods `getName` and `toString` are using the inherited variables, but otherwise the extension is orthogonal.

In unit `org.eclipse.debug.internal.ui.views.launch.LaunchView`, in class `LaunchViewLabelProvider` there are 6 inherited variables and 2 methods. $LCOM_{l,p}$ is 0.0 and $LCOM_{i,p}$ is 0.66. One method accesses two inherited variables and the other method accesses three inherited variables. These variables are used through `super`-calls. This is an example of a case where indirect calculation is needed. Inheritance is used by extending the methods. $LCOM_{a,p}$ value is relatively good, but it is still far from the

minimal value. This is because the base class is relatively large, and it is not realistic to assume that the child class would extend all of its features.

In package `org.eclipse.jdt.internal.corext.refactoring.rename`, in class `RenameVirtualMethodProcessor`, there are 8 inherited variables and 4 methods. $LCOM_{l,p}$ is 0.58 and $LCOM_{i,p}$ is 0.63. Two of the methods are accessors, but the other two use most of the inherited and locally defined variables. Also, $LCOM_{l,i}$ is 1.0, so $LCOM_{a,a}$ is high, 0.88. These two methods are also examples of methods that are extended by using super-calls. This is an example of a normal class that is quite tightly related to its base class.

The class `java.util.Collections.SynchronizedList` has 2 inherited variables and 12 local methods. $LCOM_{l,p}$ is 0.0 and $LCOM_{i,p}$ is 0.58. The inherited variable `mutex` is used by almost all methods, but the other inherited variable `c` always points to the same object as the local variable `list`. This new variable is made to simplify handling of types. The variable `mutex` is used to implement synchronization, which is an example of a cross-cutting feature. This kind of cross-cutting feature connects the elements of classes even though there might not be any logical connections. This is similar to methods like `toString` that use all of the variables, even though the variables might be completely unrelated.

$LCOM_{i,p}$ measures the usage of the base class. There are several ways in which inheritance can be used, but it seems that $LCOM_{i,p}$ cannot be used to differentiate between them. The child class can use the parent class in many ways. Another possible situation is that the child class might be strongly connected to its immediate parent, but that parent has implemented an orthogonal extension. Other ways to measure orthogonality would be to measure the proportion of methods that use any of the instance variables, or the proportion of instance variables that are used. If there are methods like `toString`, the proportion of variables that are used would be high, and the proportion of methods using inherited variables would be low. On the other hand, if the parent is used for signalling or synchronization, the proportion of methods would be high, and the amount of variables used would be low. If the class extends the base class using features of the base class, both proportions should be high.

Strong connectedness (0.0-0.5)

There are only 922 classes having $LCOM_{i,p}$ less than 0.5, and most of these classes are very small.

The class `osgi.framework.adaptor.core.BundleFile.DirBundleFile` in Eclipse has one inherited variable and 6 local methods. Four of these methods use the inherited variable, and the remaining two are empty stub methods. $LCOM_{i,p}$ is 0.33. The parent class defines a variable that contains

the data of the object.

The class `org.apache.tools.ant.taskdefs.compilers.JavacExternal` has 22 inherited variables and one locally defined method. This method uses all variables, mostly by calling the inherited methods. The parent class describes a data set, and the child implements an operation over that set.

In these classes, the parent and the child are very strongly connected. To characterize what these classes are like, we need to find out what requires the parent and the child to be separate classes. In the example cases, the parent had data, and the child had functional code. There were also other reasonable extensions of the base classes.

Discussion

The first observation is that the inheritance relation is often quite orthogonal. One reason for this is that several methods are accessors, and they are only needed for locally defined variables. Also, there are often a great number of inherited variables, which makes $LCOM_{i,p}$ higher. There is much variation in the way inheritance is used in different cases, so it is hard to say anything definite based only on the value of $LCOM_{i,p}$.

Table 2.7 shows the distribution of classes in JDK and Eclipse for $LCOM_{l,p}$ and $LCOM_{i,p}$. $LCOM_{i,p}$ is usually bigger than $LCOM_{l,p}$. The range 0.8-1.0 is the most common for $LCOM_{i,p}$ on all values of $LCOM_{l,p}$.

Based on $LCOM_{i,p}$, the inheritance relations can be classified as follows: If $LCOM_{i,p}$ is 1.0, the inheriting class is orthogonal to the base class. If $LCOM_{i,p}$ is greater than $LCOM_{l,p}$ and $LCOM_{i,i}$, the inheritance relation is not tight. If $LCOM_{i,p}$ is similar to $LCOM_{l,p}$ and $LCOM_{i,i}$, the inheritance relation is tight. This comparison might be unfair, because there are usually many more inherited variables than local ones.

There are certain problems with the above classification. One is that it does not take into account how the base class uses the inheriting class. This problem can be addressed with the metric $LCOM_{l,i}$. Another problem is that even though the class does not seem cohesive based on $LCOM_{l,p}$ or $LCOM_{a,a}$, it might be logically cohesive. For this, one should measure the clients of the class, and see how the clients use all features of the class.

Often the connectedness is based on methods like `toString`, which might use all the variables. These methods do not implement the core functionality of the class, they support functionality similar to language run time features. This is also a problem with cohesion calculations that are based on measuring the number of connected components.

	0.0-0.1	-0.2	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0	1.0
0.0-0.1	105	8	3	8	6	12	1	0	1	1	1
0.1-0.2	19	5	4	4	8	1	3	1	2	0	3
0.2-0.3	23	2	8	5	4	5	1	5	4	0	2
0.3-0.4	32	5	10	32	5	15	8	3	6	2	3
0.4-0.5	30	3	2	16	9	19	8	5	3	0	3
0.5-0.6	110	22	19	29	27	86	15	21	15	2	10
0.6-0.7	74	21	31	59	39	89	51	30	31	7	10
0.7-0.8	110	21	32	61	37	115	76	74	49	23	8
0.8-0.9	121	47	32	87	57	231	170	134	118	29	12
0.9-1.0	154	30	38	104	80	238	209	231	273	108	27
1.0	639	39	72	103	62	294	140	103	115	17	56

Table 2.7: Distribution of classes according to their $LCOM_{l,p}$ and $LCOM_{i,p}$ values. Columns stand for $LCOM_{l,p}$ ranges and and rows stand for $LCOM_{i,p}$ ranges.

2.6.3 Abstract classes

$LCOM_{l,i}$ measures how the base class uses the inheriting class. If cohesion is high with local variables and inherited methods, the base class is used like an abstract class, and the child class fills the holes left in the implementation (reification inheritance [67]). Otherwise, the methods overridden by the child class are not called by the base class.

From the 13265 classes of the sample set that have locally defined instance variables, 3857 got $LCOM_{l,i}$ values below 1.0. This means that most of the classes do not have any virtual methods that would be called by methods in the parent class, or they have that kind of methods, but they do not use any local variables. A very large part of values below 1.0 indicate very low cohesion. This is no surprise, since reification inheritance is just one of the 13 ways described in [67]. The mean for $LCOM_{l,i}$ is 0.96 for the applicable cases. Table 2.8 shows that the classes not having inherited methods using locally defined variables are usually quite small. $LCOM_{l,i}$ only correlates with $LCOM_{a,a}$, because $LCOM_{l,i}$ is often a large component in $LCOM_{a,a}$. Otherwise $LCOM_{l,i}$ does not depend on the size of the class, but more on its type.

Eclipse class `jdt.internal.ui.workingsets.SelectWorkingSetAction` has $LCOM_{l,i}$ value of 0.97. There is one public method `run`, which uses all of the locally defined variables. In the parent class, there is a method `runWithEvent`, which by default calls the method `run`. There are also lots of similar classes inheriting from the same base class, where there are no locally defined variables. Then the method `run` does not use any variables

Range	IV	V	IM	M	$LCOM_{a,a}$	$LCOM_{l,p}$	$LCOM_{i,p}$
1.0	3.1	4.2	15.2	7.1	0.79	0.40	0.25
0.9-1.0	19.1	5.3	65.1	8.9	0.88	0.49	0.81
0.8-0.9	7.9	3.5	17.8	5.6	0.77	0.37	0.57
0.0-0.8	4.1	2.9	19.5	6.7	0.58	0.26	0.43
Cor.	0.00	0.06	0.01	0.03	0.32	0.11	0.01

Table 2.8: Average values of selected attributes for different $LCOM_{l,i}$ ranges, and the correlations of these attributes with $LCOM_{l,i}$.

and $LCOM_{l,i}$ becomes trivial. The method `run` can be thought as a function, and then the method `runWithEvent` can be thought as a higher-order function, which gets the function `run` as an argument via inheritance.

In `org.eclipse.jdt.internal.ui.refactoring.ExtractTempWizard`, in class `ExtractTempInputPage`, $LCOM_{l,i}$ is 0.98. The method `setVisible` uses two of the local variables through protected methods. $LCOM_{a,a}$ is 0.90. For understanding this class, it would be useful to have a tool that shows which variables a method uses, and which methods it calls.

The class `org.apache.tools.ant.taskdefs.XmlProperty` has $LCOM_{l,i}$ 0.96. $LCOM_{a,a}$ is 0.89. The parent class has method `perform` that calls the locally defined method `execute`. This method overrides a stub method.

The class `SynthTreeUI` in package `javax.swing.plaf.synth` has $LCOM_{l,i}$ 0.97. $LCOM_{a,a}$ is 0.90. Its methods `installUI` and `uninstallUI` use almost all variables, including the locally defined ones via protected methods `installDefaults` and `uninstallDefaults`.

The class `team.internal.ccvs.core.CVSMergeSubscriber` in Eclipse has $LCOM_{l,i}$ 0.87. $LCOM_{a,a}$ is 0.82. Many of the inherited methods use variables `baseTree` and `remoteTree` via accessor methods that implement abstract methods. The values are high because there are very many inherited methods.

The class `org.eclipse.jdt.internal.compiler.ast.FieldReference` has $LCOM_{l,i}$ 0.89. $LCOM_{a,a}$ is 0.99, mostly because there are 343 inherited variables. Locally defined variables are accessed by calling methods that override stub methods defined in the parent classes. Perhaps this kind of utility methods should be implemented as static methods, because that would give more information about what they are like. Similar possibility is making them final.

The Eclipse class `HistoryAction.ImageImageDescriptor` in package `pde.internal.ui.view` has $LCOM_{l,i}$ 0.33. $LCOM_{a,a}$ value was calculated to be 0.25. There is only one variable in the class, and it is used through an abstract method.

In the `jdt.internal.corext.refactoring.typeconstraints.typesets` package of Eclipse, class `SuperTypesSet` has $\text{LCOM}_{\text{l,i}}$ 0.57. $\text{LCOM}_{\text{a,a}}$ is 0.67. Locally defined variables are accessed by simple abstract methods.

The class `DebugGraphicsFilter` in package `javax.swing` has $\text{LCOM}_{\text{l,i}}$ value 0.75. $\text{LCOM}_{\text{a,a}}$ is 0.92. The only local instance variable `color` is used via the only locally defined method, which implements a kind of function.

Discussion

Abstract classes can be much more elegant than their alternatives. Otherwise the code of methods in abstract classes would have to be written for each class, leading to longer code. This kind of use is related to mixins. On the other hand, abstract classes are harder to understand, and thus they are more prone to errors, because the control can move from a parent class to a child class, and vice versa.

It might be useful to use the technique implemented in the measure tool to output a model of a class, which can then be used to understand how the class works.

$\text{LCOM}_{\text{l,i}}$ measures a certain kind of code reuse based on using virtual methods. There are two ways to use virtual methods when implementing inherited classes. The virtual methods might implement accessors to data, or they might implement a function. The former is similar to implementing a parent class that has abstract data, and then implementing operations on that data. In both cases the data and operations are separated to different classes. The parent/child relationship is not so clear on these classes. If multiple inheritance is supported, these cases could be implemented using two parent classes where one defines data, and another defines the operations.

In the latter case, the child class implements a function, and the parent class implements a kind of higher order function based on that function. This could also be implemented using inner classes to make it more similar to functional programming.

The value given by $\text{LCOM}_{\text{l,i}}$ is usually very high, so this cannot be used to classify the classes, it just tells if abstract methods are used at all. Because there are usually a lot of inherited methods, this metric explains also why the values of $\text{LCOM}_{\text{a,a}}$ are usually so high.

2.7 Disconnected cohesion graphs

In Section 1.7.5 we studied how to measure the cohesion of graphs. These metrics can be used for evaluating the cohesion of classes. The metrics for cohesion graphs are highly related to each other, but they give somewhat different results. There does not seem to be any logical argument why one should prefer one of them over the others.

Range	IV	V	IM	M	$LCOM_{l,p}$	$LCOM_{i,p}$	$LCOM_{l,i}$
1.0	0.6	4.7	5.3	0.7	0.09	0.04	0.95
0.9-1.0	22.1	4.3	79.7	7.4	0.40	0.73	0.73
0.8-0.9	5.4	3.6	18.5	6.0	0.39	0.46	0.79
0.7-0.8	3.3	2.5	14.0	5.4	0.29	0.31	0.65
0.6-0.7	1.6	2.4	9.9	5.9	0.23	0.24	0.72
0.5-0.6	0.8	1.8	8.5	6.0	0.16	0.20	0.68
0.0-0.5	0.6	1.9	8.5	12.4	0.08	0.08	0.60
Cor.	0.34	0.20	0.32	0.08	0.34	0.54	0.32

Table 2.9: Average values of selected attributes for different $LCOM_{a,a}$ ranges, and the correlations of these attributes with $LCOM_{a,a}$.

The number of (separate connected) components in the cohesion graph can be used as the most coarse-grained definition of internal cohesion. A large number of components can be considered as an indicator of low cohesion, because there are no internal relations between the components. If there is only one component, then a member of a class is at least related to some other member of the class and transitively to every other member. This is quite a weak form of cohesion, and therefore, more fine-grained cohesion metrics have been defined.

If having several components in the cohesion graph is not caused by design errors, it is unlikely that other cohesion metrics could be used to find design errors. At least it is possible to find a way to eliminate valid causes of components from the calculation. In this way, a better metric can be defined that encodes more design knowledge.

The analysis below will show that most of the components come from unused variables, or from methods that do not use any variables. In the following, we establish that simple methods that use no inherited or locally defined instance variables can usually be justified, so they can be ignored from the calculation of cohesion. On the other hand, presence of unused variables is more likely to be due to suboptimal design.

One perspective to modularity related metrics is to consider the worst possible case of modularity, i.e. the case of just one module. The cohesion graph would then have a component that represents the code that is actually used in the program, and the rest of the cohesion graph is related to unused code.

2.7.1 Disconnected classes

The cohesion graph of a *disconnected* class has several components. If not otherwise mentioned, components caused by simple methods are ignored in

	p	i	a
l	$LCOM_{l,p}$ The traditional way to measure cohesion. Good for measuring orthogonal extensions.	$LCOM_{l,i}$ Measures the virtual method usage.	$LCOM_{l,a}$ Redundant by $LCOM_{l,p}$ and $LCOM_{l,i}$.
i	$LCOM_{i,p}$ Measures the usage of inherited variables.	$LCOM_{i,i}$ Measures the parent class.	$LCOM_{i,a}$ Redundant by $LCOM_{i,i}$ and $LCOM_{i,p}$.
a	$LCOM_{a,p}$ Redundant by $LCOM_{l,p}$ and $LCOM_{l,i}$.	$LCOM_{a,i}$ Redundant by $LCOM_{i,i}$ and $LCOM_{l,i}$.	$LCOM_{a,a}$ Good for measuring classes that use the features of the parent class.

Table 2.10: Summary of inheritance and LCOM. Subscript a means all instance variables or methods, i means all inherited variables or methods, p means locally defined public methods and l means locally defined variables.

the following, and the inherited part is considered to be included in the class. The left panel of Table 2.11 shows the distribution of the number of components for the cohesion graphs of $LCOM_{l,pns}$ and $LCOM_{a,ans}$. The table reveals that surprisingly many classes have more than one component. When inherited properties are included in the classes, they typically have more components, because in general, the number of components grows when going lower in the inheritance hierarchy. The largest number of components for $LCOM_{l,pns}$ and $LCOM_{a,ans}$ is 1188 in class `ORBUtilSystemException` in `com.sun.corba.se.impl.logging`, and 71 in class `GtkWidgetClass` in package `org.eclipse.swt.internal.gtk`, respectively. The first case is caused by methods using inherited variables. In the second case, the class has 71 instance variables.

As an example, consider the class `javax.swing.JDialog` with 16 components. Most of the variables and methods form a single component.

Components	$NOC_{l,pns}$	$NOC_{a,ans}$
0	5340	3008
1	9149	7838
2	2863	3610
3	1286	2103
4	742	1462
5	432	1064
6	283	435
7	198	288
8	138	216
9	93	134
10-	508	879

Ratio	Local	Inherited
0.0-0.1	216	119
0.1-0.2	458	344
0.2-0.3	665	499
0.3-0.4	722	556
0.4-0.5	288	401
0.5-0.6	1477	1271
0.6-0.7	693	1458
0.7-0.8	564	1430
0.8-0.9	862	1844
0.9-1.0	604	2270
1.0	14488	10845

Table 2.11: Distribution of classes according to their number of components and size proportion of the largest component from the class. Simple methods are ignored. The local case means the cohesion graph used when calculating $LCOM_{l,pns}$ and the inherited case is the cohesion graph that is used with $LCOM_{a,ans}$.

Some instance variables are used only by private serialization methods, and these are not measured in our definition. Analysis shows that the variables `warningString` and `temporaryLostComponent` form their own components with their corresponding accessors. The variables `modalComp` and `modalAppContext` are only used by reflection in the package `javax.swing`. The variable `modalExcluded` only seems to be used by constructors. The variable `focusCycleRoot` is unused, because all the methods using it have been overridden. It is clear that identifying components helps in finding potential problems in the classes.

One clear problem in using NOC as a problem indicator is that methods like `toString` might artificially force that the class consists of a single component. Calculation formulas based on the connectedness of the graph, like CBMC, or ignoring these kind of special methods might be useful.

As another example, consider class `ElementListSelectionDialog` from package `org.eclipse.ui.dialogs`. The class consists of one connected component. It inherits from `AbstractElementListSelectionDialog`, which has 9 components. That in turn inherits from `SelectionStatusDialog`, which has 4 components (inherited from `SelectionDialog`). The class `ElementListSelectionDialog` combines all components from the classes it inherits to a single component with a method that uses all variables. This shows that even if a class makes orthogonal extensions, like `SelectionDialog` made to `org.eclipse.jface.dialogs.Dialog`, inheriting classes might make

the extensions non-orthogonal. This is because the components might be conceptually connected, but this cannot be measured by inspecting the class automatically.

Causes for components

One cause for the formation of a component is an unused variable. A variable might also become unused, if all the methods using it are overridden. This is clearly a problem in the implementation and design of classes.

The second cause for a disconnected cohesion graph is that a class is meant to be used by inheritance. In this case the class should be declared abstract.

The third cause is that a class defines two or more disconnected features. This is the most complex case to analyze. It is possible that the motivation for disconnected designs comes from clients of the classes. Often these objects are used as records with plain accessors by the clients. Such a design is clearly not cohesive, but it might be convenient to group several features like this. A problem would arise if the clients include code, that could be implemented as a method in the class. However, extracting such a client code as a method for the record class might not be feasible, because the code could also use intensively some other data. It is common that a large class has an instance variable that is used by accessor methods only. Similarly, unused variables form disconnected components. In these cases, it should be checked, whether the component is actually used by any of the clients. Finally, it is possible that two or more components are so unrelated that the class should be split into two or more classes.

To separate the cases where there are unused variables from the cases where the class is used as a record type, one can inspect how large is the largest component in the class. The right panel of Table 2.11 reveals the ratio of the size of the largest component with respect to the size of the whole usage graph. The table indicates that the largest component is often very large meaning that a large number of components does not directly imply a bad LCOM value.

Components and inheritance

By using components, we propose a concrete classification of classes. There are four categories. *Fillers* are classes that have less components in the cohesion graph with inheritance than in the local cohesion graph. There are 1655 fillers in the sample set. These classes override some methods, and the base class combines their features. *Normally extended* classes are classes that have the same number of components as their base class. *Adders* are classes that add more components to the base class. *Fixers* are classes that

have less components than their base class. There are 5368 adders and 1286 fixers in the sample. The remaining 14219 classes are normally extended. Of the adders, 2980 add just one more component.

We give some typical examples of these categories. The Eclipe class `jdt.core.Member` is an example of fillers. It consists of 8 components when examined locally. It includes no locally defined variables, but all of the methods are not ignored as simple methods because they use inherited variables. If the inherited methods are counted in, there is only one component. Another filler is the class `org.apache.tools.ant.listener.AnsiColorLogger`. It has a protected method `printMessage` that uses all 6 locally defined instance variables. There are no public methods in the class, so the class has 6 components in the local case. The protected method is called from the parent class. The fillers are examples of classes with inherited methods using locally defined variables via overridden methods.

An example of fixers is `javax.swing.tree.VariableHeightLayoutCache`. This concrete class extends an abstract class, which has several components. Most fixers are similar to that.

When considering the formation of components, the most important cases are the adders. Classes of this type are responsible for all disconnected classes. They often have no inherited variables, because `java.lang.Object` has no variables. An example of this kind of class is `java.util.Currency`. An adder class usually has two or three related variables, but it only provides the accessor methods, and thus the objects of such a class are pairs, or more generally tuples. The following examples show how it is possible to find all causes of components by inspecting adders without having to inspect classes that do not cause them.

Adders are (partially) orthogonal extensions of their parent classes. For example the class `javax.management.monitor.MonitorNotification` inherits from the class `javax.management.Notification`. The child class just includes getters and setters for the instance variables it defines. These notification classes are examples of classes that are used as records. The needs of the client for these classes are so various that there are no meaningful methods that should be included in the class.

2.8 TCC and other alternatives to LCOM

TCC (Tight Class Cohesion) calculates the cohesion as the ratio of method pairs sharing one or more variable and the number of all possible method pairs. TCC and CBMC are a kind of opposites, because in TCC, if one variable is used by all methods, it gives the best cohesion, and in CBMC, if one method uses all variables, but other methods use just one variable, it gives low cohesion. Because of this it seems motivated to define met-

ric rTCC, which calculates pairs of variables that are both used by some method. From these definitions, it follows that rTCC ignores methods that use one or zero instance variables, and TCC ignores unused variables.

On the basis of correlation calculations, it is seen that TCC and rTCC do not correlate much with the number of variables. On the other hand, rTCC has strong correlation with the number of components. The advantage of rTCC over LCOM and TCC is that its correlation is weak with the number of variables, but its correlation is strong with the number of components.

TCC gives value 1.0 for 687 and rTCC for 2103 classes. When TCC is 1.0, all methods are in different components, and when rTCC is 1.0, each variable is in its own component. One reason that there are more cases where rTCC is 1.0 is that a variable often has two accessors, a getter and a setter.

Maximal TCC

There are 687 classes with maximal TCC value 1.0. These classes mostly have only methods that use a single variable and each method uses a different variable. There are also some methods that use two or more variables that are not used by any other methods. These classes usually do not have very many variables (2.6 on average). They tend to inherit directly from `java.lang.Object`, and they have no inherited instance variables.

Maximal rTCC

There are 2103 classes with rTCC 1.0. In these classes, methods only use one variable at most. If the class has only simple methods, TCC is 0.0 but rTCC is 1.0. The average number of variables is 4.4 and each variable forms its own component.

Class `java.net.URL` gives rTCC 0.98, and TCC 0.69. One of the variables is used in several methods, but only one method uses more than one variables. The class consists of 23 methods, when inherited methods are included.

2.8.1 Comparison of LCOM and TCC

Considering classes, their average number of variables begins to grow when LCOM values are above 0.5. First, they grow slowly with increasing LCOM values, but in the end they grow very quickly. In TCC, the number of values is generally higher, with peaks at 0.45-0.5 and 0.7-0.75. The first peak is caused by Eclipse's AST classes, and the other by Swing.

The number of components grows quite linearly when TCC grows. With LCOM, the number of components starts growing after 0.3.

When there are lots of variables, at least one of these variables might be such that all methods use it. Then the value of TCC is 0.0, and because of this, LCOM and TCC do not correlate much. Otherwise they correlate strongly.

LCOM and TCC correlate least, when considering all variables, including inherited ones, and just local methods. This can be explained by the following reasoning: When there are lots of inherited variables that are never used, LCOM gives low cohesion, but if there is a local variable that is used much, TCC gives high cohesion. On the other hand, if there are for example two methods, and they both use a different variable, TCC gives the lowest cohesion value, but LCOM gives 0.5.

The relation of TCC with the number of methods is similar to the relation of LCOM with the number of methods. LCOM correlates strongly with the number of variables, especially when trivial methods are ignored. On the other hand, TCC does not correlate at all with this number. This is because if there are a lot of variables, there might be a variable that is used by all methods, and then the other variables do not make the cohesion lower. So, LCOM and TCC values do not correlate when all inherited variables have been taken into account, because then there are lots of variables. Also when there are less methods, the correlation is weaker.

Based on the above considerations, TCC and rTCC should be preferred to LCOM, because they correlate better with the number of components, and less with the size of the class. On the other hand, the value given by LCOM gives more information, and its results are easier to understand. But the most important case for cohesion values is the situation where there are no multiple components, and this will be investigated below.

2.8.2 Components and LCOM

It is interesting to compare LCOM and TCC to the number of components as indicators of problems in class design. It is evident that LCOM and TCC are more useful for classes that have only one component. If there are several components, LCOM is not so good measure, but then the class probably needs to be inspected anyway.

Most big classes have several components, due to unused variables or unrelated feature sets. In the sample set, there are 10845 classes that consist of one component, when all inherited variables and methods are considered. Now if we look at $\mathrm{LCOM}_{a,a}$ (in Table 2.12) and keep only the classes that have one component, there are only 64 classes in the 0.9-1.0 range, and 293 in the 0.8-0.9 range. This is an example of the strong correlation between LCOM and the number of components.

One can generate random cohesion graphs based on the number of variables, the number of methods and LCOM of a class. The table for this

	0.0-0.1	-0.2	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0
1	6359	295	421	840	644	774	660	496	293	64
2	328	18	33	79	237	1018	645	631	446	171
3	166	2	1	13	11	101	928	453	331	96
4	141	0	0	3	0	15	50	536	438	277
5	61	0	0	3	0	5	32	100	469	393
6	58	0	0	0	0	0	9	42	220	106
7	44	0	1	0	0	1	4	4	149	85
8	36	0	1	0	0	0	0	9	110	59
9	33	0	0	0	0	2	1	3	63	32
10-	132	0	0	0	1	1	1	7	67	670

Table 2.12: Number of classes with a certain number of components in the cohesion graph (as given by rows) and a certain range of $LCOM_{a,a}$ (as given by columns).

randomized graph is very similar to Table 2.12. It seems like the expected value of the number of components can be determined straightforwardly from the number of methods, the number of variables, and LCOM (or more simply, the total number of used instance variables in methods). This is surprising, because it could be assumed that classes tend to consist of one component. The interpretation of this result is that internal cohesion is not a guiding principle in object oriented design.

An example of a class with high $LCOM_{a,a}$ (0.87) and one component is `ListDialog` in package `org.eclipse.ui.dialogs`. It has 25 instance variables, and 27 methods. Of these methods, 20 use only one variable, but two methods (`open` and `create`) use all the instance variables. Another example is Eclipse class `Browser` in package `org.eclipse.swt.browser` with $LCOM_{a,a}$ value of 0.93. It has 431 methods, and one of them (the method `createCOMInterfaces`) uses 48 of the 54 instance variables. This method is only called by the constructor.

There is often a single method that uses all variables, and keeps the class as a single component. These methods can either implement the core functionality of the class, or they can for example be a method (like `toString`) which artificially connects the class into one component. LCOM values would still be high, though. To get more precise measurement of cohesion, a connectedness based methods like CBMC could be used.

2.8.3 Connectedness of cohesion graphs

Often, the connectedness of a component depends on one method or variable. If this method or variable is not a part of the core functionality of the class,

it binds the class together in a way that does not imply cohesion. The idea of connectedness analysis is to remove nodes (methods or instance variables) from a cohesion graph, and see if it is still connected.

In most cases, removing a variable splits the class into components easier than removing methods, for example the accessors of the variable become components. For this reason, we propose another connectedness analysis which removes only methods.

One feasible way to implement connectedness analysis could be implemented by removing one or two elements from the graph, and seeing how many components there are then. When removing variables, one might also remove the methods that become trivial when they are removed.

For example the class `java.util.HashMap` has one component, if methods that use no instance variables are ignored. If the instance variable `entrySet` is removed, the methods that use only the variable `entrySet` become their own components. On the other hand, if the method `clone` is removed, the variables `keySet` and `values` form their own components. These variables are connected by inner classes.

An example of a one-component class is `java.awt.JobAttributes`. This class has very high cohesion. Class `java.awt.Button` has 5 components, but only one of them is large. The large component has low cohesion, because the method `addNotifyListener` is the only method that uses instance variable `nativeInLightFixer`.

2.8.4 Components and TCC

Classes where there are several components can always be considered non-cohesive. Therefore TCC and rTCC are most interesting for one-component classes.

TCC

With TCC, the range 0.6-0.7 has 248 classes, the range 0.7-0.8 has 76 classes, and the range 0.8-0.9 has 75 classes.

The class `java.security.cert.PKIXBuilderParameters` has 13 variables and 30 non-simple methods. TCC is 0.88. One method (`toString`) uses all variables, one uses two, and others use just one variable. This is a good example of a phenomenon that can be detected using TCC but not by using the number of components.

A interesting example is in `com.sun.org.apache.xpath.internal.axes`, class `ChildTestIterator`, where there are in total 25 variables. Two methods use 17 variables. This shows that single-component classes do not always have methods that use all variables. TCC is 0.82.

rTCC

With rTCC, the ranges 0.4-0.5, 0.5-0.6, 0.6-0.7 include 121, 71 and 29 classes, respectively. If there is one component, it seems that quite often there is a method that uses all of the variables: rTCC is 0.0-0.1 in 6674 cases from the total of 7838 cases.

In class `com.sun.org.apache.xml.internal.utils.SAXSourceLocator`, there are has 5 variables and 8 nontrivial methods. rTCC is 0.60, TCC is 0.64 and LCOM is 0.7. Of the methods, 4 access only one variable, and the other four access two variables. The first four are setters in the base class, and the other four are getters that are redefined in the child class. The setters bind the class together using variable `m_locator`. So there seems to exist a genuinely loose cohesion in the class.

The class `debug.internal.ui.actions.RemoveAllExpressionsAction` of Eclipse has 7 variables and 8 methods that use variables. TCC is 0.43, rTCC is 0.61, and LCOM is 0.75. The methods access at most 4 variables. The class seems to have several related functions, which all need their own variables.

The class `org.eclipse.update.core.model.FeatureModel` has 31 variables and 65 methods. The methods use at most 10 different variables. rTCC is 0.76. The class is bound together by variable `readOnly`. This variable is used to implement write protection. It is an example of a variable that binds the class together artificially.

Conclusions

TCC and rTCC usually give quite good values for classes which consist of one component, and if the values of the two metrics are weak, they are usually signs of weak cohesion. It was found out that if there is one variable or method that is related to all other variables or methods, rTCC and TCC give the perfect value 0.0. The two measures have the same weaknesses relating to connectedness but used together, they can detect shallow graphs.

2.8.5 Other alternatives

A natural way to measure the cohesion of a graph that has not yet been considered is LCC. LCC is lowest, if the class has only one component. LCC is highest if each method has its own component. In this sense LCC is the same as the number of components. One improvement of LCC over NOC is that if there is one method with its own component, LCC gives a lower value than if there were two components with equal size. A problem that still remains is that both methods give the highest value for a case which is not usually bad design (c.f. data class).

2.9 Elements of Cohesion

To understand, what internal cohesion might tell about classes, one has to inspect its constituents. These are

- Methods with no used instance variables.
- Methods with one used instance variable.
- Relations between instance variables used by methods.
- Internal calls between methods.

To analyze causes for disjoint components and low internal cohesion in general, it is useful to inspect methods with no accessed fields and methods with only one accessed field. To complete the classification, we classify the roles and relations of the fields in the methods that use more fields.

In the present work, it is one of the most interesting findings that this kind of classification can be assisted by modern software analysis tools. Methods can be divided into groups based on simple syntactic properties such as which language constructs are used in the methods. Taking n such properties the methods can be divided into 2^n groups which can then be systematically inspected to find different concepts. In the process, also design rules and design anomalies can be found.

Some language design issues make this kind of classification harder. For example using static fields to help in the classification is difficult, because in Java, there are two uses for static fields; as global variables, or for defining enumerations.

Languages such as Java might have operationally similar, but a bit different constructs. For example a method can be called in Java in the following ways:

- Normal dynamic call.
- Dynamic call from this-variable.
- Super-method call from this-variable.
- Static method call.
- Constructor call.
- Super-constructor call.
- This-constructor call.

This shows that the conceptual complexity of objects cannot be hidden by the language.

2.9.1 Methods using no fields

In the test material, there are 1286 methods that do not use any fields. These methods are in 401 classes. Of these classes, 130 do not have any locally defined fields. It is clear that the classes that have both methods using fields and methods not using fields are most interesting.

Of the methods using no fields, 737 are overloaded. This shows that these methods are often used via dynamic binding. In fact, if these methods are not used via dynamic binding, they are quite pointless. Another feature of methods using no fields is that they are almost always very simple: 1231 of them have neither looping nor branching.

Of all methods, there are 374 stub methods with no functionality at all, not even a return value. There are 1470 methods that only include one expression. Most of the methods including only one expression are getters.

Of the methods that use no internal variables, 254 have no expressions and 348 have only one expression. The single expression might be a constant value for a class that is accessed via dynamic binding. This is a kind of a variable that depends on the class of the object. This kinds of variables cannot be implemented as static variables, because each class in the inheritance hierarchy shares the inherited variables with other classes. This shows that a class as a run-time entity is different from a class as a language construct.

Based on the properties above, the methods can be classified into four groups. The first group consists of the methods with no functionality, and are used via overloading. There are 185 such methods, and they are all the same. In addition there are 48 such methods that are not overloaded. These are stub constructors, unimplemented methods, or methods that could be overloaded, but this is not done in the test material.

A peculiar case are methods that return this-variable, an example of this is `String.toString`. There were 6 such methods. It is actually surprising that there are no more methods of this kind, because identity operations should be pervasive when modelling domains.

The second group consists of overloaded methods with one expression. There are 242 such methods. As discussed before, these methods can be thought to be constant fields, that are related to the class of the object. They can be used for example to tell what kind of capabilities the class has. Even if these represent constants, the field or property might be a variable in other classes in the hierarchy. One possibility is that the method implements an abstract factory pattern.

The third group has 97 non-overloaded methods with one expression. These can for example return static fields. Some are deprecated methods and others are stub methods that throw exceptions. Most are methods that could be overloaded so they do not differ from the second group.

Methods of the fourth group contain several expressions. They imple-

ment the FUNCTION design pattern. In this pattern, an object of a class is used like first-class functions in functional programming.

As a summary, methods using no fields can be classified to:

1. Stubs: either do nothing or throw an exception.
2. Abstract factory methods.
3. Function design pattern methods. This group does not include all methods related to the functional pattern, because e.g. a class might represent a family of functions and then the method would access an instance variable.
4. Methods only using this-variable.
5. Constant property that is related to a class. Might be a normal property in other classes.

If a class implements a functional pattern and has unrelated instance variables or methods, this can be a signal of low cohesion. Otherwise, these methods seem to be necessary and parts of sound design.

2.9.2 Usage of this-variable

Even if a method does not use any field, it can still use the `this`-variable. One possible use is `this`-passing, where `this`-variable is given as an argument for a method. Another use is testing for equivalence with `this`. It should be possible to classify the uses for `this`-variable.

Of the methods that use no fields, 191 call `this`-variable. There are the following possibilities: check the type of `this`, use `this`-passing, or call a method from `this` that uses no fields. In general, `this`-variable is most of the time used for accessing its fields.

For usage of `this`-variable, there are the same cases as for arguments

- Pass the variable forward.
- Call a method from the variable.
- Just return the argument.
- Store the argument to a variable. This operation is not useful for `this`-variable.

This-passing is often used with complex recursive relationships. For example an object might be a part of a more complex pattern, which determines its properties. Another possibility is that a class A has a static method, that takes an object of the class A as an argument. Then some

methods call the static method with this-passing. Perhaps it is thought that this way of calling is more efficient than using normal methods.

As a conclusion, only this-passing seems to be problematic. In our opinion, the methods should operate on the fields of the class, and not on the whole object. However, this-passing is not a true case of not using instance variables and it is therefore unrelated to cohesion.

2.9.3 Methods using one field

The sample set includes 2743 methods that use exactly one field from the class they are in. These are the most common methods and so they need to be inspected carefully.

A large part of these methods does not have any branching or looping (in fact only 653 methods have looping or branching control structure). A majority of the methods are not overloaded: 636 methods are used with overloading. As a more fine-grained approach to complexity of these methods, the number of expressions in each method was counted, see Table 2.13.

The methods can be classified to categories:

- Getters, which return one of the fields.
- Setters, which modify one field.
- Wrapper method for another method using only one field.
- Wrapper method calling a method from a field.

Note that there can also be getters and setters that get or set several fields.

For a wrapper method calling another method from a field, there are two possibilities: either a call chain is avoided, or the class is a wrapper for the field. When a call chain is avoided, there are two conceptually different cases: the field is a part of the object, or the field is a relation to another object.

More generally, a getter might be a function on the field. For example this kind of method can check for a property of a field. An example is checking the size of a collection. In Java, there are several collections that implement an isEmpty-method in the same way as it is already implemented in an abstract base class of collections. This indicates a need for an analysis for determining whether a method can be moved up in the inheritance hierarchy.

Methods using one field reveal problems concerning composition of classes. For example there are useless wrappers of basic data structures (probably implemented before genericity was added to Java). Another example is the container pattern for a small set represented as an integer. Another problem with fields in general is that there is a major conceptual difference between a field representing an attribute or a relation.

Number of expressions	Methods
1	857
2	318
3	321
4	351
5	108
6 or more	788
Total number of methods	2743

Table 2.13: Analysis of the methods using exactly one field of its class, showing how many expressions these methods use.

Getters

There are several different kinds of getters:

- Simple getters.
- Getters with consistency check.
- Lazy creation of a field.
- Cloning getter: returns a clone of the object contained by the field. These are useful when the object enforces an additional invariant on the contained object. Otherwise the client of the object could change the object into an inconsistent state.
- Conversion from a representation to another. For example a field that represents an angle could return the value in degrees, while the internal representation would be in radians.

If lazy creation of fields is applied, one may have different kind of creation for inherited classes. A problem here is the typing, because the type could be different but it should not be exposed outside. The listing in Table 2.14 shows how this pattern can be implemented in Java.

Regarding consistency checks, we measured how many null-tests there are in the source in total. There are 7779 tests for equality using operator ==. Of these, 2929 test for null. In addition, of 6306 tests for inequality with operator !=, 3986 test for null. Another typical test is for testing if an object has a given dynamic type and checking for array bounds.

A problem related to getters is that sometimes getters and fields are used confusingly: A class may have methods that access a field directly, and also another method that accesses the field via the getter method. It should be defined in the programming style, which way is used for accessing the field in a particular situation.

```
class LazyField<T> {
  private Factory<T> fac;
  private T val = null;
  public LazyField(Factory<T> f) {
    fac = f;
  }
  public T get() {
    if (val == null) val = fac.create();
    return val;
  }
}
```

Table 2.14: Lazy creation of fields in Java.

Setters

Setters can be classified similarly to getters:

- Simple setters.
- Setters that check the argument value.
- Set constant, toggle, increment.
- Setter to clone or reference.
- Setter may set several fields or construct a new object from arguments.
- Setter converts the argument from one representation into another.

Many setter methods are used for implementing the observable design pattern.

Language features such as generics and enumerations can be used to achieve great improvements in design of programs. Perhaps similar solutions could be found for the design problems involving getters and setters. For example traits and other methods of composing classes could be used to simplify the implementation of the observer pattern. A trait $A; B$ would mean a class with methods $a(x) = A.a(x); B.b(x)$. A problem is that recursive relations might become confusing traits, similarly to normal inheritance.

Wrappers

Of the methods that are not clearly getters or setters, and still use only one field, most are wrappers, which call some simple operations from a class. Wrappers have similar considerations as getters and setters. For example a wrapper might make some small additional conversion, just like a getter.

A common pattern is to have a container as member, and then operations to add or remove from this container. There are now two possibilities: the class augments the container with some value, or the class has several different containers.

If a class implements a complex wrapper for one of its fields, the class can be thought to have many roles, and therefore low cohesion. This wrapper could be separated into its own class.

If a complex operation is performed on a field, why would it be in this class that only contains the field? We can assume that large methods are more likely to indicate design problems.

More formally, the following design can be considered anomalous: a method is large, the method uses one field, and the class of the method has more than one field. In general, the less instance variables a large class uses, the more suspicious its design is. The correlation between the number of used fields and the size of methods can be used to confirm this design to be anomalous. A part of this correlation is caused automatically, because at least one expression must be added for each access. This kind of method can also operate on the arguments. An example of this kind of wrapper that contains functionality that should clearly be moved into the wrapped class is `JProgressBar.getPercentComplete`.

If these cases would be fixed so that functionality is moved into their correct place, it would not improve the cohesion of the original class. However, the cohesion of the class, where the functionality belongs, might become higher. This shows that while LCOM cannot be used to find problematic cases, it can perhaps be used as a design heuristic to find out which refactorings are good.

Other observations

It was observed that many objects which were contained in the fields were directly accessible using getters. Less fields were available for setting. However, it is often possible to modify the objects that are returned by the getters. It would be interesting to determine, in how many cases, the underlying objects are really encapsulated.

2.9.4 Field relations

We next investigate what kind of relations can exists between two or more fields. It turns out that a surprisingly large part of the inspected relations between fields can be explained by just few programming patterns. A method that uses two fields can be thought to describe an *interaction* between the fields. Most common interactions are related to properties of objects and states of objects. A problematic, but very common cause of interactions is

inlining classes.

Inlining and optimization

Performance related optimizations often cause relations between fields. A problem is that it might be hard to know if the trade-off between performance and simplicity is good, or if it was even considered.

A common optimization is that some simple class is "inlined" inside another class instead of using the simple class. If the class is very simple, it might sometimes actually be easier to inline than to use the class. It would be important to develop tools to detect this kind of inlining.

For example, 2d-coordinates are sometimes inlined so that there are fields such as x and y (then there are of course the actual implementations of points and such). One reason is that there is no good language support for vectors in Java.

A more low-level example is a buffer that has been implemented as an array. Sometimes an optimized map can be defined so that some keys are stored in fields while others are stored in a normal map.

An example of another kind of optimization is a method that initializes a field, and also a related optimization field. For example a container might have a field that contains the size of the container.

Another related problem in Java is that for values, there is no null-object. Because of this, an extra field is needed to signal that a field is not set. Also the validity of a value of a field might depend on the state of the object in some other way. The natural solution for value types not having null would be using the corresponding object wrappers (boxed values). Perhaps this is not done because of performance reasons. Or perhaps it is because boxed values were cumbersome to use in older versions of Java. Again, it is hard to tell if this kind of design choices have not been documented in the source code.

Properties

One cause for interactions are the *properties* of objects. At first one could think that the fields contain the properties, since they are mappings from object identifiers to values. In many cases, the association of properties to objects can be much more complicated.

A property of an object can be defined by some other object. In this case the object is a part of a larger system, that is the actual holder of the property. Also there can be two or more alternatives for where the property can be found.

One possibility is that the data might have an inheritance-like pattern, where default values are searched from some kind of prototype object.

In addition to properties, also the functionality might have alternatives (delegation).

These examples show that handling the properties of objects is more complex than adding fields to classes. The solution is to separate property handling patterns into their own separate data structures. One challenge is having the same status for data structures such as trees and graphs as sets, lists and maps have now.

State of objects

In addition to properties, an object can be thought to have a *state*. The properties can be thought to belong to the state, too. State dependant functionality can be detected by checking if a field involves control flow.

If a state is changed, there might be some associated object that is signalled: for example an observer or a "mirror". Another possibility is that when making a state change, the cache should be cleared, or the state of contained objects should be changed.

The value of a field might be related to another field. In this case, this field represents the state of the other field.

Miscellaneous relationships

In some cases, a field remembers an attribute or previous value of another field. This corresponds to the follower role of local variables. It can be thought that each possible role of a variable there is a corresponding role for fields. For example a method can simply return a commonly needed expression of two or more attributes. This is similar to the role of temporary variables.

In concurrent object and field access, there are often objects that are related to synchronization. This is a natural relation between two objects. On the other hand, this also is a programming pattern that can be tackled with for example aspect oriented programming.

One pattern comes from functional programming. If a function has two parameters, one can make another function that has one of these parameters fixed. Translated into objects, one field would represent the function object, and another would represent the fixed parameter.

When inspecting classes, it can be noticed that classes have parameter roles for the method parameters. Two different methods in a class can have parameters with the same role. In these cases, usually the same name is used for both methods. When a parameter role can have several types, an extra field might be needed for decoding the parameter.

Conclusions

When inspecting methods that use two fields, it often seems that the system could be redesigned so that the methods could involve only one field.

It can be assumed that a common case is that the context of the object has two other objects that must be used to implement the necessary functionality.

To classify the roles of methods more formally, one can use for each involved variable the following aspects:

- Contained objects.
- Related objects.
- Derived objects (objects that are created based on the object).

Contained and related objects are found using fields. One can also consider for example derived objects of contained objects etc. This kind of chains should not be very complex, because they are considered in a context of one method. To classify a method, the data and control flow in relation to these considered objects should be inspected.

What seems to be difficult in programming is finding the correct level of abstraction. In particular, too low-level concepts are often used. Because only the interface of the class is used by the clients of the class, bad design decisions internal to the class are not shown outside the class. Therefore the pressure to remove these problems is small. The emphasis of design can be thought to be a feature of the waterfall process model, where the design is thought to be fixed during implementation. In agile models, implementation is thought to influence design more.

2.10 Conclusions

The key idea for this chapter is that cohesion measures how well the class is implemented regarding the principle of encapsulation. However, encapsulation is not something that would always be desirable. The actual desired principle would be that the class has a simple interface. Encapsulation follows if the internal representation is more complex than the interface. The ideal case would be that both internal representation and the interface are simple. In this case encapsulation is not needed.

We can now see that the cohesion of a class cannot always be seen when inspecting the class in isolation. Because of this, we need to model the relations of classes. When inspecting relations between classes, it is also necessary to consider coupling. If the cohesion depends on some other class, this might be a case of tight coupling. Then, the class with low cohesion

can be seen as a utility class. In modern software engineering, using design patterns and software architectures creates these kinds of simple utility classes. To solve these problems we need to implement more general metrics to measure modularity.

The worst case for modularity should be that there is only one class. We tested a model, where all classes were combined to one, and measured its cohesion. The measures could not find differences with it and normal big classes. This gives more support to our argument that internal cohesion metrics are not enough, some other metrics are needed, too.

It was found out that the cohesion measures depend on the size of the classes, most strongly on the number of variables. One problem with this is that the variables should be hidden from the interface as an implementation detail. We were not able to remove the dependency by using simple statistical methods. To solve this we might need invariants or some similar help to determine how the class should be used. It would then be possible to measure modularity even before the classes have been implemented.

The metrics we investigated seem to give a good idea of what the internal cohesion of classes is like, even though there still remain some problems. LCOM is the most basic metric and it also seems that the expected value of other metrics depends on the value of LCOM. The number of components is an important measure and perhaps best captures the idea on internal cohesion. Of other metrics, rTCC seems to give most useful results.

One problem was that when using the concept of cohesion from modules to measure the cohesion of classes, it is not simple to know how the inheritance should be handled. Because of the inheritance, a class can be thought to be an extendible module. When considering the class as a module, all inherited components should be included. But also the nature of the extension can be useful to determine. It would be particularly helpful, if there were a way to determine, if the extension actually follows from an is-a relation, or should composition have been used instead.

In the definitions, it was useful to define the cohesion graph, which was then used by the different calculation methods. Variating methods and variables was not that useful, because the useful variations were quite easy to find.

For measuring cohesion, the call graph seems to be the most useful model of the program. There are some ways to make the call graph more precise that should to be investigated. One of these is points-to analysis. Also more analysis is needed to compare the direct and indirect instance variable use.

We proposed an approach where the class is divided into four different parts: the inherited part, the locally defined part, and two parts to connect them. This division was useful to analyze the relationship of cohesion and inheritance. In particular, it was observed that the part where inherited methods use locally defined variables, is often empty.

So, how could the cohesion metrics be used? The easiest metrics to use are the ones based on the number of components. If a class has several components, it needs to be checked whether the components are logically connected. If the class has unused variables or small components, it must be analyzed if they are used by some other classes, or if they could be replaced somehow.

The number of components must be calculated in a way, that does not cause false positive findings. Because of this, inheritance must be considered. Indirect computation is not relevant, because connectedness analysis is transitive anyways. Method calls must be resolved though.

There is a problem when using the internal cohesion metrics for searching suspicious classes: It can be thought that the most suspect case would be that a class has two cohesive components. However the cohesion measures usually give worst results when each field has its own disconnected component. These classes are mostly so called data classes and they are usually not examples of bad design, although they can be examples of evolving designs.

If the classes have only one component, perhaps LCOM, TCC or rTCC can be used. To find out if they are useful, we would have to use them in software projects. rTCC seems to be a good measure, because it correlates well with the number of components and is quite independent of the number of variables.

In the present chapter, two main problems of cohesion metrics were identified. First, the metrics cannot measure the logical cohesion of classes. To solve this problem, we will introduce the concept of external cohesion, where methods are replaced with client classes. The second problem is that cohesion cannot be used for software quality improvement. For this we will measure how much refactorings can be used to improve values of the metrics. For example a class consisting of two cohesive components could be split into two parts, producing maximal cohesion improvement. Another example is moving a large method that uses just single field to some better location.

Chapter 3

External Cohesion Metrics

3.1 Introduction

External cohesion is investigated in this chapter. Instead of internal relationships, we consider external relationships, where two methods are related, if they have a common user.

To better understand the relationship between internal and external cohesion, one has to consider how the classes are understood by the developers. There are three alternatives corresponding to different views into classes. First, a class can be considered as a separate entity with a meaning independent of its context. This is the *internal view* of the class, which is used when writing the class. Second, when a class A is used by another class B, the other class B uses its *client view* of the class A. Each client of the class A has its own view of the class A. Different clients might for example assign different roles to the class they are using. The third view is the *global view*, which is the combination of the two other views. This kind of view is used in high-level design and code reviews. This view is also natural for metrics. *Client-based metrics* are metrics that use the client and internal views to measure classes. It can be easily seen that client-based metrics are useful. They are needed for complex refactorings that affect several classes. For example, in Move Member [43] refactoring, a member is moved between two classes that are not related by inheritance. Metrics that use only the internal view cannot reliably assist in such refactorings. On the other hand, metrics that only give a value for a package or a project make it too hard to find the reasons for a bad value.

As discussed in the previous chapter, the problem with existing object-oriented cohesion metrics is that they only use the internal view to the class, and therefore give a limited account of the cohesion of classes.

If two different clients of a class have very different client views of the class A they are using, it can be assumed that the class A does not implement

a maximally cohesive set of features. To detect this kind of lack of cohesion, we propose the metric LCIC (Lack of Cohesion In Clients) that measures, how coherently the clients of a class use it. If all clients use all features of the class A, then it has clients that have a similar view of the class A. Otherwise, the class A might have unnecessary features, or it might implement several different concepts.

During software evolution, new features are added into a class. This can make the class uncohesive. As an example, consider a class that represents a point in three-dimensional space. If an application also has a need for two-dimensional points, it can reuse the 3D point class for this. But in this case, the point class would have unnecessary features. It would be useful to detect this kind of usage of classes.

A class is not always used in the same way by all its clients: It then has different roles for different clients. For example, a communication channel might have roles for input and output. Different roles can be specified as interfaces in object-oriented languages such as Java. Because of this, we have designed our metric LCIC to support classes with several interfaces.

Consider again the point class example. If there are methods drawing 2D graphics into 3D space, there might be classes that use only X- and Y-coordinates of the point. These classes should use the point through a 2D interface. Then they can also operate on true 2D points.

The organization of this chapter is as follows. Related works are presented in Section 3.2. In Sections 3.3 and 3.4 we define the LCIC metric for Java programs. Experimental results are presented in Section 3.5. The relation of the proposed metric to different design patterns and refactorings is analyzed in Sections 3.6 and 3.7. LCIC metric is an example of client-based metrics – in Section 3.8 we discuss possible variations for LCIC. Conclusions are drawn in Section 3.9.

This Chapter is based on the publications [61], [60] and [62].

3.2 Related Work

Ott and Thuss [74] present slice-based metrics for measuring functional cohesion in procedural programs. In these metrics, the measured modules are sliced based on their output values i.e. only statements that are necessary for generating a specific output value are left in the program. The sliced modules are then compared. This is similar to the metric proposed in the present work, where only variables that are used by clients remain in the class.

KABA [89] uses static and dynamic analysis about the clients of classes to refactor class hierarchies. For each object creation site, a new class is created. The used members of these classes are arranged into a new class

hierarchy discovered by concept analysis. The advantage of KABA, compared to a metrics based approach, is that it can suggest and perform valid refactorings. On the other hand, KABA can only suggest moving members in a refactoring hierarchy. In addition to this, KABA currently applies only to bytecode.

Star diagrams [72] are program visualizations that can be used to find accessor chains and other problems with duplicated code. Because star diagrams are based on parse trees, the accessor chains must have almost exactly same code. Our detection of accessor chains does not depend on the shape of code at all, only on the semantics.

Distance based cohesion [88] by Simon et al. has been used to identify refactorings. Distance of methods and variables is calculated based on their similarity. Similarity is based on the methods and variables they use. The most important difference between this and the present work is that our work applies to classes, whereas the distance based cohesion applies to methods.

Misic [69, 70] argues that the client-based approach is superior to the traditional approach, and suggests that the cohesion of a component should be measured by using the formula of LCOM* with clients and features of the component. Misic gives the metric in a general fashion and does not fix the definitions of components, clients or features. The client-based approach for cohesion measurement is also used by Ponisio and Nierstrasz [77]. Unlike Misic and us, they use the formula of TCC instead of LCOM*. Ponisio and Nierstrasz have implemented their metric for calculating the cohesion of packages.

The automatic detection of design patterns has been concentrated on detecting the instances of design patterns in existing programs. This kind of tools usually are able to detect factory methods, see for example [54].

In a book on applying object-oriented metrics by Lanza and Marinescu [56], only one of the defined metrics (CC) examines the clients of the measured class. The cohesion metric used there is TCC.

3.3 External cohesion metric LCIC

In object-oriented modelling, a class should represent a single concept from the problem domain. Internal cohesion metrics try to measure internally, how well a module represents a single concept. The assumption behind internal cohesion metrics is that a module represents a single concept, if the parts of the module are closely interconnected internally. In LCIC, it is attempted to measure, whether a client views the used module as a single concept. Further, it is attempted to combine the 'views' of all clients as a metric value for the module. The intuition behind this is that a module represents a single concept, if the parts of a module are connected (e.g. used)

by each client. If clients do not view the module as a single concept, it can be assumed that the module is not well designed with respect to cohesion.

To define the new cohesion metric LCIC (Lack of Cohesion In Clients), the concepts of modules, clients, and parts of a module are defined first. It is also necessary to handle different roles of the modules, and define the formula that calculates the numeric values of the metric from the model.

As modules, we consider *flattened classes*, which include all locally defined members (fields and methods) and additionally all inherited members. An alternative definition would be to consider only locally defined members. This alternative is however unsatisfactory, because the functionality of a class cannot be understood without the inherited members.

There are two obvious ways of interpreting what it means that a class is a client of another class. The first is to consider only direct usages, where a client uses the methods or instance variables of another class, and another is to consider all usages transitively. The disadvantage of the latter approach is that there would be several irrelevant clients, which just use the class via some other class as a hidden implementation detail. This approach is also less efficient, because the sets of variables used by methods are then much larger. The former approach might also cause problems, because a client, which directly uses an object of a class, might pass that object to another class, which further accesses it. This problem occurs rarely in practice, so we choose to primarily consider direct usage.

Next we need to define, what are the parts of a class. A class provides services to other classes. Methods of a class represent its services. A disadvantage of this approach is that there are often utility methods that are not needed by all clients, for example a class representing sets might have several methods, but most clients need only methods for adding an element and testing if an element belongs to the set. One possibility would be to mark the core methods, or use something like abstract fields [67], but this would require too much manual support. Another, more promising possibility is to consider fields of the class as its parts. If a field is not used by the clients of the class, it is an unused resource for each instance of that class. For these reasons, we choose instance variables to be the features that are measured.

A class can have different roles for different kinds of clients. An interface corresponds to a possible role of a class. In the definition of LCIC, only the used interfaces are taken into account. If a class has features that cannot be accessed by the interfaces a client uses, the clients' inability does not cause any penalty to the value of LCIC.

To calculate a numeric LCIC value for a target class, we use the same approach as in LCOM* [49, 15]. The values of LCIC range from 0 to 1, where 1 represents the worst coherence. For each client, we calculate how many features it does not use from the class compared to the total number of

features of the class that can be accessed through the interfaces available for the client. LCIC is the average of these ratios when considering all clients of a given class. By doing this, it is easy to compare LCIC to various internal cohesion metrics.

3.4 Definition of LCIC

In this section, we give a precise definition of the LCIC metric for Java. First, a model of the programs is given, and it is described how the model can be constructed from Java programs. LCIC is finally defined based on this model.

3.4.1 A model of programs

The model of a program includes a set of *members* (instance methods and fields) $\mathfrak{M}$ and a set of *classes* $\mathfrak{C} \subseteq \wp(\mathfrak{M})$, where $\wp(\mathfrak{M})$ is the power set of $\mathfrak{M}$. A class is modeled by the set of its members. Because we use the flattened versions of classes, all classes are pair-wise disjoint sets. For the example program of Figure 3.1, the set $\mathfrak{M}$ has three elements that represent fields; variable `x` in the class `A`, the inherited variable `x` in class `B`, and variable `y` in class `B`. Set $\mathfrak{M}$ contains also all the inherited methods for the classes, so for the example program (ignoring Java's implicit superclass `Object`), the set is

$$\mathfrak{M} = \{\texttt{Test.test}, \texttt{A.x}, \texttt{A.setX}, \texttt{A.getX}, \texttt{B.x}, \\ \texttt{B.super.setX}, \texttt{B.getX}, \texttt{B.getX}, \texttt{B.y}, \texttt{B.square}\}$$

A *view* for a class is modeled as a subset of that class. Each superclass and interface of the class in the program induces a view in the model. Note that because a client can use a class through several interfaces, all views in the model need not be representable by an interface or type in the program code. The set of all possible views is

$$\mathfrak{V} = \{\mathfrak{v} \subseteq \mathfrak{M} \mid \mathfrak{c} \in \mathfrak{C}, \mathfrak{v} \subseteq \mathfrak{c}\}$$

In the example program, class `B` has a view that does not contain the method `square()` and field `y`. This view corresponds to the class `B` seen through its supertype `A`. The *maximal view* for a member $\mathfrak{m}$ is the class which contains $\mathfrak{m}$.

To complete the model of programs, we define a *call relation* $\text{Call}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v})$ to hold, if a method $\mathfrak{m}$ directly accesses a member $\mathfrak{m}'$ using a view $\mathfrak{v}$, where $\mathfrak{m}'$ is a called method or an accessed field.

```
public class A {
  protected int x = 2;
  public void setX(int xx) {
    x = xx;
  }
  public int getX() {
    return x;
  }
}
public class B extends A {
  private int y = 1;
  public int square() {
    y = getX()*getX();
    return y;
  }
  public void setX(int xx) {
    System.out.println("Setting " + xx);
    super.setX(xx);
  }
}
class Test {
  public static void test(A a) {
    if (a == null) return;
    a.setX(12);
    a.setX(-12);
    a.setX(2*a.getX());
  }
}
```

Figure 3.1: An example program.

3.4.2 Constructing the model

To construct a model from a Java program, a set of *Java types* Type and a set of *Java member signatures* MSig are needed. Let $Super(\mathfrak{t}, \mathfrak{v}) \subseteq \text{Type} \times \wp(\mathfrak{M})$ be a supertype relation which holds, iff the type $\mathfrak{t}$ induces the view $\mathfrak{v}$ on some class. Further, let $Sig(\mathfrak{s}, \mathfrak{m}) \subseteq \text{MSig} \times \mathfrak{M}$ be a signature relation which holds, iff the signature $\mathfrak{s}$ can be used to access the member $\mathfrak{m}$. Additionally, let Sig_s be another signature relation which represents `super`-calls.

These sets and relations can be easily extracted by a method similar to the type checking procedure of Java programs. In the example program (Figure 3.1), there are two types `A` and `B`. There are signatures corresponding to each defined member. The type `A` induces another view to the class `B`. Signatures that are defined for the class `A` can be used to access members from `B`, too.

To calculate the relation Call, all expressions from the method bodies of each member method $\mathfrak{m}$ are inspected. The inherited methods have to be inspected for each class. $\text{Call}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v})$ is defined as follows:

- If the method body for $\mathfrak{m}$ contains a constructor call or a static call of a method $\mathfrak{m}'$, $\text{Call}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v})$ holds with the maximal view $\mathfrak{v}$ for $\mathfrak{m}'$.

- If the method body for $\mathfrak{m}$ contains an access $\mathfrak{f} \in \text{MSig}$ applied to a `this`-object, then there is one member $\mathfrak{f}' \in \mathfrak{v}$ such that $Sig(\mathfrak{f}, \mathfrak{f}')$ holds, where $\mathfrak{v}$ is the maximal view for $\mathfrak{m}$ and $\mathfrak{f}'$. In this case, $\text{Call}(\mathfrak{m}, \mathfrak{f}', \mathfrak{v})$ holds. Calls from `super` are handled in the same way, except that the relation Sig_s is used instead. `super`- and `this`-statements in constructors are handled as in normal methods. In the example program of Figure 3.1, all instance variable accesses are from `this`. Because the type of `this` is known at the time of analysis, the classes `A` and `B` are not clients of each other.

- Assume that the method body contains a dynamic call with signature $\mathfrak{f} \in \text{MSig}$, which is accessed from an expression with type $\mathfrak{t} \in \text{Type}$. Then consider the set $\mathfrak{A}$ of all members, which are contained by views induced by the type $\mathfrak{t}$, and can be called via the signature $\mathfrak{f}$, that is,

  $$\mathfrak{A} = \{\mathfrak{m} \in \mathfrak{M} \mid Sig(\mathfrak{f}, \mathfrak{m}) \wedge \exists \mathfrak{v}.\, Super(\mathfrak{t}, \mathfrak{v}) \wedge \mathfrak{m} \in \mathfrak{v}\}$$

  For each member $\mathfrak{a}_i$ in $\mathfrak{A}$, we calculate a view $\mathfrak{v}_i$ based on the type of the receiver variable. Then the relation $\text{Call}(\mathfrak{m}, \mathfrak{a}_i, \mathfrak{v}_i)$ holds. In the example program, the method `Test.test` accesses the methods `A.getX`, `A.setX`, `B.getX` and `B.setX` via views induced by the type `A`.

- Otherwise, the relation $\text{Call}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v})$ does not hold.

Following these rules, the Call-relation for the program in Figure 3.1 can be calculated:

$$\begin{aligned}\text{Call} = \{ & \\ & (\texttt{A.getX}, \texttt{A.x}, \mathfrak{A}), (\texttt{A.setX}, \texttt{A.x}, \mathfrak{A}), (\texttt{B.getX}, \texttt{B.x}, \mathfrak{B}), \\ & (\texttt{B.setX}, \texttt{B.super.setX}, \mathfrak{B}), (\texttt{B.square}, \texttt{B.y}, \mathfrak{B}), \\ & (\texttt{B.square}, \texttt{B.getX}, \mathfrak{B}), (\texttt{Test.test}, \texttt{A.setX}, \mathfrak{A}), \\ & (\texttt{Test.test}, \texttt{A.getX}, \mathfrak{A}), (\texttt{Test.test}, \texttt{B.setX}, \mathfrak{B}_{\mathfrak{A}}), \\ & (\texttt{Test.test}, \texttt{B.getX}, \mathfrak{B}_{\mathfrak{A}})\}\end{aligned}$$

The views $\mathfrak{A}$, $\mathfrak{B}$ and $\mathfrak{B}_{\mathfrak{A}}$ are:

$$\begin{aligned}\mathfrak{A} &= \{\texttt{A.getX}, \texttt{A.x}, \texttt{A.setX}\} \\ \mathfrak{B} &= \{\texttt{B.getX}, \texttt{B.x}, \texttt{B.setX}, \\ &\quad\ \texttt{B.super.setX}, \texttt{B.square}, \texttt{B.y}\} \\ \mathfrak{B}_{\mathfrak{A}} &= \{\texttt{B.getX}, \texttt{B.setX}\}\end{aligned}$$

3.4.3 Definition of the LCIC metric

For a class $\mathfrak{c}$ and methods $\mathfrak{m} \in \mathfrak{c}$, an internal transitive call relation is defined as

$$\begin{aligned}&\text{Call}_{\mathfrak{c}}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v}) = \text{Call}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v}) \vee \\ &\quad \exists \mathfrak{m}'' \in \mathfrak{c}.\text{Call}(\mathfrak{m}, \mathfrak{m}'', \mathfrak{c}) \wedge \text{Call}_{\mathfrak{c}}(\mathfrak{m}'', \mathfrak{m}', \mathfrak{v})\end{aligned}$$

In other words, $\text{Call}_{\mathfrak{c}}(\mathfrak{m}, \mathfrak{m}', \mathfrak{v})$ holds, if there is a call chain of methods inside class $\mathfrak{c}$, where the first method is $\mathfrak{m}$ and the last method accesses member $\mathfrak{m}'$ via view $\mathfrak{v}$. In the example program, $\text{Call}_{\mathfrak{c}}$ is the same as Call, except that the methods `B.square` and `B.setX` are related to `B.x`.

The set $\mathfrak{F} \subseteq \mathfrak{M}$ is the set of fields. We define class $\mathfrak{c}$ to be a *client* for field $\mathfrak{f} \in \mathfrak{F}$ *via* view $\mathfrak{v}$ when

$$\begin{aligned}\text{Uses}(\mathfrak{c}, \mathfrak{f}, \mathfrak{v}) \quad = \quad & \exists \mathfrak{d} \in \mathfrak{C}, \mathfrak{m}' \in \mathfrak{c}, \mathfrak{m}'' \in \mathfrak{d}.(\mathfrak{f} \in \mathfrak{d} \wedge \text{Call}(\mathfrak{m}', \mathfrak{f}, \mathfrak{v})) \vee \\ & (\mathfrak{f} \in \mathfrak{d} \wedge \text{Call}(\mathfrak{m}', \mathfrak{m}'', \mathfrak{v}) \wedge \text{Call}_{\mathfrak{d}}(\mathfrak{m}'', \mathfrak{f}, \mathfrak{s}))\end{aligned}$$

The left term of the disjunction denotes a direct usage of a variable, and the term on the right denotes a usage via a member method call. As shorthands, denote $\text{Uses}(\mathfrak{c}, \mathfrak{f}) = \exists \mathfrak{v}.\text{Uses}(\mathfrak{c}, \mathfrak{f}, \mathfrak{v})$ and $\text{Uses}(\mathfrak{c}, \mathfrak{c}') = \exists \mathfrak{f} \in \mathfrak{c}'.\text{Uses}(\mathfrak{c}, \mathfrak{f})$. In the example, the class `Test` is the client for the field `x` in `A` and `B`. Also, the classes `A` and `B` are clients for themselves because they use their own fields:

$$\begin{aligned}\text{Uses} \quad = \quad & \{(\mathfrak{A}, \texttt{A.x}, \mathfrak{A}), (\mathfrak{B}, \texttt{B.x}, \mathfrak{B}), (\mathfrak{B}, \texttt{B.y}, \mathfrak{B}), \\ & (\texttt{Test}, \texttt{A.x}, \mathfrak{A}), (\texttt{Test}, \texttt{B.x}, \mathfrak{B}_{\mathfrak{A}})\}\end{aligned}$$

The *client view* of a class $\mathfrak{d}$ for a client $\mathfrak{c}$ is the union of views via which the class $\mathfrak{c}$ uses members of $\mathfrak{d}$:

$$\begin{aligned}\text{View}(\mathfrak{c}, \mathfrak{d}) \quad &= \quad \bigcup_{\mathfrak{f} \in \mathfrak{d}, \mathfrak{v} \subseteq \mathfrak{d}, \text{Uses}(\mathfrak{c}, \mathfrak{f}, \mathfrak{v})} \mathfrak{v} \\ \text{clients}(\mathfrak{d}) \quad &= \quad \{\mathfrak{c} \in \mathfrak{C} \mid \text{Uses}(\mathfrak{c}, \mathfrak{d})\} \\ \text{accessible}_{\mathfrak{c}}(\mathfrak{d}) \quad &= \quad \{\mathfrak{f} \in \mathfrak{d} \mid \text{Uses}(\text{View}(\mathfrak{c}, \mathfrak{d}), \mathfrak{f})\}\end{aligned}$$

For example, the client view of `B` for `Test` includes only the members that are accessible by using the methods in `A`. Using the client view, the set $\text{accessible}_{\mathfrak{c}}(\mathfrak{d})$ is defined as the set of fields accessible from $\mathfrak{d}$ by the interface available for $\mathfrak{c}$. The set $\text{clients}(\mathfrak{c})$ is the set of clients for the class $\mathfrak{c}$.

Now, *lack of cohesion per client* is the ratio of the features that a client $\mathfrak{c}$ does not use from the class $\mathfrak{d}$ to the features accessible by the client views of the class $\mathfrak{d}$ for the client $\mathfrak{c}$:

$$LCIC(\mathfrak{c}, \mathfrak{d}) = 1 - \frac{|\{\mathfrak{f} \in \mathfrak{d} \mid \text{Uses}(\mathfrak{c}, \mathfrak{f})\}|}{|\{\mathfrak{f} \in \mathfrak{d} \mid \text{accessible}_{\mathfrak{c}}(\mathfrak{d})\}|}$$

Finally, LCIC is the average lack of cohesion of clients for the class $\mathfrak{d}$:

$$LCIC(\mathfrak{d}) = \frac{\sum_{\mathfrak{c} \in \mathfrak{C}, \text{Uses}(\mathfrak{c}, \mathfrak{d})} LCIC(\mathfrak{c}, \mathfrak{d})}{|\text{clients}(\mathfrak{d})|}$$

If a class has no clients, LCIC is zero, when the class has no members, and otherwise one. All classes in the Figure 3.1 have LCIC value 1.

3.4.4 Call chains

In the following we give a different formulation of the use relation, to make presenting new variations of LCIC metric easier. A *call chain* is a sequence of member-view-pairs $\Sigma = \mathfrak{M} \times \mathfrak{V}$. As an abuse of notation, $\mathfrak{m}$ denotes the set $\{(\mathfrak{m}, \mathfrak{v}) \mid \mathfrak{v} \subseteq \max(\mathfrak{m})\}$ and $\mathfrak{c}$ denotes the set $\{(\mathfrak{m}, \mathfrak{v}) \mid \mathfrak{v} \subseteq \mathfrak{c}, \mathfrak{m} \in \mathfrak{c}\}$, where $\max(\mathfrak{m})$ is the maximal view for member $\mathfrak{m}$. We define the path $w \cdot (\mathfrak{m}, \mathfrak{v})$ to be valid for the program if the call chain finally accesses member $\mathfrak{m}$ via view $\mathfrak{v}$ after w:

$$\begin{aligned} \text{valid}(\epsilon) &= \text{true} \\ \text{valid}((\mathfrak{m}, \mathfrak{v})) &= \mathfrak{m} \in \mathfrak{v} \\ \text{valid}(w \cdot (\mathfrak{m}', \mathfrak{v}') \cdot (\mathfrak{m}, \mathfrak{v})) &= \text{valid}(w \cdot (\mathfrak{m}', \mathfrak{v}')) \wedge \text{Call}(\mathfrak{m}', \mathfrak{m}, \mathfrak{v}). \end{aligned}$$

We extend the predicate valid for sets of sequences:

$$\text{valid}(A) = \exists w \in A.\text{valid}(w).$$

For example, the set of internal paths for class $\mathfrak{c}$ is $\mathfrak{c}^*$. The set of internal field accesses is $\mathfrak{c}^* \cdot (\mathfrak{c} \cap \mathfrak{F}) = \mathfrak{c}^* \cap (\Sigma^* \cdot \mathfrak{F})$. The set of internal field accesses using view $\mathfrak{v}$ is $\mathfrak{c}^* \cap ((\mathfrak{c}, \mathfrak{v}) \cdot \Sigma^*) \cap (\Sigma^* \cdot \mathfrak{F})$. Internal paths for a field $\mathfrak{f}$ can be defined as $\text{internal}_{\mathfrak{v}}(\mathfrak{f}) = (\max(\mathfrak{f})^* \cdot \mathfrak{f} \cap (\max(\mathfrak{f}), \mathfrak{v}) \cdot \Sigma^*)$. Now the Uses relation can be defined as

$$\text{Uses}(\mathfrak{c}, \mathfrak{f}, \mathfrak{v}) = \text{valid}(\mathfrak{c} \cdot \text{internal}_{\mathfrak{v}}(\mathfrak{f})).$$

Note that if there are three classes $\mathfrak{a}, \mathfrak{b}, \mathfrak{c} \in \mathfrak{C}$, $\text{valid}(\mathfrak{a} \cdot \mathfrak{b} \cdot \mathfrak{c})$ means that there is a chain of methods where $\mathfrak{a}$ calls $\mathfrak{c}$ via $\mathfrak{b}$, not only that $\mathfrak{a}$ accesses $\mathfrak{b}$ and $\mathfrak{b}$ accesses $\mathfrak{c}$.

It is possible to define a language $A \subseteq \Sigma^*$ to be the language such that $\text{valid}(B)$ is equivalent to $A \cap B \neq \emptyset$.

3.4.5 Axioms of cohesion measures

Briand et al. [15] have proposed a set of axioms for theoretical validation of cohesion measures. In their model, a class has a set of *relationships*. This set is *maximal*, if new relationships cannot be added into a class. In the case of LCIC, a relationship can be defined to be a triple $(\mathfrak{a}, \mathfrak{b}, \mathfrak{c})$, where $\mathfrak{a}$ and $\mathfrak{b}$ are fields in the class, and $\mathfrak{c}$ is a client that uses both fields $\mathfrak{a}$ and $\mathfrak{b}$.

There are four axioms for the lack of cohesion:

1. Normalization: The values of the metric are in a normalized range $[0, \mathrm{Max}]$. The definition of LCIC is not normalized, because every client uses at least one field of the measured class. We can define normalized versions of lack of cohesion per client and LCIC with range $[0, 1]$:

$$\begin{aligned} LCIC'(\mathfrak{c}, \mathfrak{d}) &= \frac{|View(\mathfrak{c},\mathfrak{d})|}{|View(\mathfrak{c},\mathfrak{d})|-1} LCIC(\mathfrak{c}, \mathfrak{d}) \\ LCIC'(\mathfrak{d}) &= \frac{\sum_{\mathfrak{c}\in\mathfrak{C},\mathrm{Uses}(\mathfrak{c},\mathfrak{d})} LCIC'(\mathfrak{c},\mathfrak{d})}{|\{\mathfrak{c}\in\mathfrak{C}|\mathrm{Uses}(\mathfrak{c},\mathfrak{d})\}|} \end{aligned}$$

2. Minimal and maximal values: If the set of relationships inside a class is maximal, the lack of coherence of the class is zero. If the set of relationships of a class is minimal, the lack of coherence value is maximal. This axiom holds for LCIC', if classes without fields are given LCIC' value 1.

3. Monotonicity: If a relationship is added into a class, the lack of cohesion does not increase. This property does not hold for LCIC, because adding a new client that uses only a bit of the class increases the number of relationships, but increases LCIC.

4. Merging unconnected classes. If two unrelated classes $\mathfrak{a}$ and $\mathfrak{b}$ are combined into class $\mathfrak{c}$, the lack of cohesion of $\mathfrak{c}$ is not smaller than the lack of cohesion of $\mathfrak{a}$ or it is not smaller than the lack of cohesion of $\mathfrak{b}$. This property holds, if the two unrelated classes do not call each other and have no common clients.

To fix the problem with monotonicity, the definition of relationship should be revised. A relationship is a triple $(\mathfrak{a}, \mathfrak{b}, \mathfrak{c})$, where $\mathfrak{a}$ and $\mathfrak{b}$ are fields in the class, and $\mathfrak{c}$ is a client that uses both fields $\mathfrak{a}$ and $\mathfrak{b}$ *or* $\mathfrak{c}$ is a class that does not use any field of the target class. The monotonicity property now holds, because if a new client is added for the class, the set of relationships becomes smaller, unless the client uses all the fields of the class.

3.5 Experimental results

To evaluate the proposed metric LCIC, experiments with LCIC and other metrics were performed. The following criteria were used to select the software suitable for these tests

- The source code should be freely available.
- The project should be of medium range. It cannot be too large, because otherwise it would take too long time to understand its logic, and it cannot be too small, either, because then the results would be trivial. So projects that were around 100000 lines were searched for.
- The code must be written in Java because the metrics tool was designed for Java.
- The software should be in mature state.

Based on these criteria, three projects were chosen: jEdit (161 KLOC, 870 classes), jabref (94 KLOC, 697 classes) and tvbrowser (151 KLOC, 856 classes). The metrics can be calculated for these libraries using the prototype of the automated tool that was described in Section 1.9. Even though the prototype is unoptimized, it can handle projects with 2000 classes in few minutes, using less than 400 Mb RAM. These results indicate that it is feasible to calculate LCIC even for large projects.

A statistical analysis was performed for the LCIC-results determined by the automated tool. Sample cases were then analyzed in detail and reasons for high LCIC values were classified.

3.5.1 Statistical analysis

Table 3.1 shows the distribution of LCIC values for the three projects in the sample. Almost half of the classes are of *trivial* type, which have only one or zero fields. In these cases, the clients use all fields, if they use some field, so LCIC does not give a meaningful value for them. In the following considerations, trivial classes are ignored. Another observation is that a large part of the classes has very small LCIC values and only few classes have a very high value.

LCIC correlates with the size of the class. For example, its correlation (Pearson product-moment correlation coefficient) with the number of inherited variables is 0.38 and with the number of inherited methods the correlation is 0.32. Correlation with the class size is not surprising, because in fact almost all metrics correlate strongly with size. LCIC correlates more clearly with LCOM (correlation 0.59) than other cohesion measures such as TCC (0.40) and the number of disjoint components in the internal cohesion

	all	jabref	jedit	tvbrowser
Trivial	1138	413	414	311
0.0 - 0.1	561	122	138	311
0.1 - 0.2	142	23	52	67
0.2 - 0.3	112	27	37	44
0.3 - 0.4	131	29	50	52
0.4 - 0.5	111	25	54	29
0.5 - 0.6	113	13	76	23
0.6 - 0.7	64	21	26	15
0.7 - 0.8	34	13	18	3
0.8 - 0.9	11	8	2	1
0.9 - 1.0	5	3	2	0
Total	2422	697	869	856

Table 3.1: LCIC values for projects. The cells show the number of classes in the projects jabref, jEdit and tvbrowser having an LCIC value in certain range. Trivial classes are classes with one or zero instance variables.

graph (0.30). It also correlates with the number of clients (0.35). It was observed that LCOM is almost always higher than LCIC. These results are promising, because one of the problems with LCOM is that it gives high values for too many classes (false positive findings).

A potential problem with LCIC is that there are not always many clients for classes. For example, classes which get good LCIC but have several components in the internal cohesion graph, tend to be used by one or two clients. Often these classes are inner classes. This is an advantage, because this kind of helper classes are not harmful. On the other hand, classes which have only one component, but high LCIC, can have problems that should be refactored.

In addition to a simple statistical analysis, individual classes were analyzed to answer the following questions:

- What is the relationship between LCIC and internal cohesion measures, in particular why is the external cohesion high, even though internally the classes give very bad cohesion values?

- Do the classes with a high LCIC value have a need for refactoring, and what kind of refactorings could be applied?

Below we show representative results.

3.5.2 Comparison of LCIC and other cohesion metrics

By inspecting a number of classes, it was discovered that LCIC has several advantages over the common internal cohesion metrics. Despite this, it turned out that LCIC cannot always replace the internal cohesion metrics. For example if a class is a main program that has no clients, LCIC cannot be applied to it.

A problem with the internal cohesion is that a method that makes the internal cohesion higher may implement functionality that is not central to the meaning of the class. The class `syntax.Token` in jEdit is an example of a class, where LCOM (0.4) is lower than LCIC (0.57). The class has 5 variables and only one method, `toString`. This method does not really tell anything about the meaning of the class. Because the clients only use the public variables of the class, LCOM and other internal cohesion metrics do not say anything useful about the cohesion of the class.

Perhaps the most visible problem of LCOM is that adding accessor methods always makes LCOM values higher, compared to classes, where variables are declared public and used directly. For example, the class `gui.HistoryText` in jEdit has several accessor methods. One of the fields is used by accessors, only. Even though this field is not connected to other fields internally, all seven clients use it via the accessor method. As a result, LCOM is 0.625 and LCIC is 0.05. Unlike LCOM, the LCIC value is not affected by accessor methods. Some proposals for cohesion calculations suggest automatic checks for detecting accessors, and ignoring the detected methods from the cohesion graph. However, it is not always easy to automatically detect accessors, because accessors might for example have checks guaranteeing that data invariants are preserved.

Data classes include a few related members and very little functionality. If variables that are used simultaneously are grouped together into a data class, LCIC suggests that this is a case of good cohesion. For example, the class `mods.PersonName` in jabref has 3 variables in 2 disjoint components. There is only one client, which uses all 3 variables. As an example, where there are several clients, the class `journal.UnabbreviatedAction` in jabref has 4 variables with 2 inherited ones in 3 components. Five clients use the class via the full interface, and five other clients use it via the interface `AbstractWorker`. All these clients use all of the features available from the interface they are using. The three examples show that the variables in a class can be connected by usage, even if there are no internal connections inside the class.

If several clients use a class in a limited way, LCIC can be used to recognize this. The class `PluginJAR` in jEdit is an example of a class, which forms one component internally, but has high LCIC (0.64). LCOM and TCC are high, too. Some clients use most of the variables, but they use

the class via different sets of methods. There are also several clients, which use only one or two variables. The class implements complex functionality and has 1697 lines of code, therefore it is a good candidate for refactoring. As another example, the class `gui.StatusBar` in jEdit has one component, and its LCIC is 0.63. Many clients use only method `setMessage`, which is called from an object received from `View.getStatus`. This suggests refactoring a new method `setMessage` to the class `View`. Possibly the method `View.getStatus` could be removed.

As a summary, LCIC has the following differences with respect to the internal cohesion metrics:

- LCIC generally returns lower values than LCOM.
- Accessors and simple methods do not cause any difference to LCIC values, but they are problematic for the internal cohesion.
- The internal cohesiveness can be caused by functionality that is not central to the functionality of the class. This is not the case with LCIC, because the functionality that is used by the client classes is the central functionality of the class.
- LCIC can be high even if a class forms one component internally.

3.5.3 Classes with high LCIC values

Even though it was shown above that LCIC does not have the problems associated with internal cohesion metrics, we still need to demonstrate the usefulness of LCIC. To be useful in improving the quality of software, high values of the metric have to indicate design problems with high probability.

We performed a small case study about using LCIC to find design problems. First, LCIC values were calculated for jedit, jabref and tvbrowser. It was noticed that the classes in tvbrowser had best LCIC values, and therefore it would be least likely to find design problems from these classes. The threshold value was set to 0.5 so that around 5 percent of the classes would have to be inspected. The classes were inspected using a tool for browsing call graphs, and the source code of classes. The reasons for high LCIC were divided into ten categories. If several different reasons were found for a class, the class was added to all these categories.

Below, we list the discovered causes for high LCIC, and show examples of this kind of classes. Some high LCIC values were not caused by design problems, but most of the time, the design of the classes with high LCIC could be improved. This indicates that LCIC has good potential to find design problems in software.

When LCIC is high, a class might implement several kinds of functionality. For example the class `AbstractCardPanel` in tvbrowser implements a

double linked list, and some additional functionality. The class has 3 fields, and its LCIC is 0.5. Clearly, the linked list functionality should be separated from the other functionality.

It was noticed that LCIC can also detect fields that are accessed by call chaining. In these cases, clients should be passed the needed part instead of what is now passed. For example, most of the clients for the tvbrowser class `devplugin.ChannelGroupImpl` use only one of the 4 fields in the class. LCIC of the class is 0.68. This is because the class is mostly used through a call chain

```
Channel.getGroup().getId()
```

The call chain can be removed by adding a function `getGroupId` into the class `Channel`.

One common reason for high LCIC is that a class includes a lot of data of which only a part is used by typical clients. Sometimes these classes are just a lump of data, and sometimes the data are better organized. Basically, these classes are used like databases, where there is a container of data, and several clients, which access the same container of data. If none of the clients needs certain data or if the data can be split into sets that only one client needs, then there is a problem in the design of the class. The class `ReminderPluginList` in tvbrowser is an example of this kind of class. The objects of this type are stored into a data structure, which is used by several clients. Additional data are needed by only a few clients. In this case, high LCIC does not indicate design problems.

LCIC of a class decreases if a new interface is associated with it. For example, the class `MutableChannelDayProgram` has methods for adding programs, getting programs by index, and iterating over the program list. This class should clearly implement the list interface of JDK. The class has LCIC value of 0.66 and it has 5 fields.

If there are fields that could be declared as static, LCIC becomes higher. For example, if a set of fields is used as an enumeration, and they are declared as final and not static, this makes LCIC high, because it is unrealistic that all clients would need all cases of an enumeration. The solution is to add the `static` modifier to the fields.

Sometimes fields are accessed by a library that is not considered when calculating LCIC. This causes high LCIC values. For example, objects of the class `simplemarkerplugin.GroupUnmarkAction` are passed as callbacks into a library. Because the library is not measured, two of the fields are left unused. The class has 3 fields and its LCIC is 0.56. This problem can be solved by adding the used libraries to the context that is used in the calculation of LCIC.

LCIC can give bad values, if objects of the class are created and configured by factory-like methods. There are two reasons for creating a new

object: The object is a helper object that is used by the creating method or a class, or the object is created by a factory-like method to be used by other objects. Perhaps an automatic check could be defined to separate these two cases. Similar to this, if an object is passed to a part object that uses it fully, this is not measured.

One of the most common reasons for bad LCIC values is that a class has fields that are used only in the initialization phase. These variables should be local variables in methods instead of fields. The fields can for example include all user interface components in a dialog.

LCIC values become worse, if a class has fields not needed by its clients. For example the class `SelectableItemList` has plenty of fields that are only used by the method `setEnabled`, which is never used by the application. However, the program might be extended so that this method would become used. In any case, GUI components already contain their children, so the fields contain redundant data. The class has 7 fields and its LCIC is 0.67. As another example, the class `BitmapHeader` has three fields that are never used by the clients. There are a total of 11 fields and the LCIC value is 0.81.

Sometimes fields and methods are used, when an inner class should have been used instead. This is a special case of uncohesive functionality. LCIC can be used to detect this problem. For example, the class `EditFilterDlg` has LCIC value of 0.47. It has 15 fields, and only one client, `SelectFilterDlg`. Most of the fields are needed by the callback function `actionPerformed`. This callback should be implemented as an inner class, and not be added as a part of the interface of the class.

As a summary, we recognize the following causes for high LCIC values:

- The class implements several different kinds of functionality.
- The class is used via a call chain.
- The class has unused variables or variables that could be declared as local or static.
- The class has some rarely used variables.
- The class is created and configured by factory methods.
- The class has a lot of information which it provides. Most clients need only a part of this information.

The results of the case study are shown in Table 3.2. Out of 42 cases, we found that 34 cases had design problems that could be fixed to improve the design of the program. This case study indicates that LCIC produces a small number of false positives and therefore, it is straightforward to use it to improve program design.

Reason	*Classes*
Call chains	11
Uncohesive functionality	7
Redundant data	5
Unused variables	5
Initialization	4
Not static	2
Interface issues	2
Rarely used data	4
Factories and passing	2
Unmeasured library	1
Total	42

Table 3.2: Classification of reasons for LCIC values that are higher than 0.5 in tvbrowser application. Some classes belong to several categories.

Threats to validity

The size of the case study was quite small. It is not certain that all the reasons for high LCIC values were found. Different software packages may have different frequencies of different causes for high LCIC. This is especially because of inheritance, also the problems can be inherited. For this reason more studies are needed to confirm the results.

The analysis was based on the relations in the code that directly lead into the observed metrics values, therefore there is not much space for subjective interpretation in the classification. On the other hand, it can be argued that for example call chains do not constitute bad design.

The above weaknesses of LCIC are removed in the Chapter 4, where more reliable experimental results are given and a model is presented that explains why the detected classes can be considered to have bad design.

3.5.4 Classes with few clients

One problem with applying LCIC are classes having only one or two clients. It is hard to get meaningful results for this kind of classes. To analyze what kind of classes have only a few clients, we inspected the jEdit application and the parts of JDK it uses. There were total of 1913 classes. Of these, 1297 classes had instance variables. Out of these, 96 classes had no clients. A part of the classes with no clients were classes like `GUIUtilities.UnixWorkaround` that are only called using a constructor. Other classes with no clients were classes that were passed to an external library that was not included in the measurement.

There were 151 classes with one or two clients. Many such classes were inner classes. For example `EditPane.CaretInfo` includes variables that are used by the `EditPane` class, but cannot be member fields, because they are related to objects associated with `EditPane`.

Another example of a class with only one client is `PropertyManager`. Its only client is the main class `jEdit`. Some functionality needed in the main class has been moved to `PropertyManager`. The main class implements a facade that can be used to access the features of `PropertyManager`.

As a summary, LCIC values for classes with one or two clients were reliable when applying the measure to the three sample projects.

3.6 Evaluation of LCIC with design patterns

There are at least two reasons, why design patterns should be taken into account when evaluating metrics. First, design patterns are considered to be exemplars of good design. Because of this, metrics should give good values for classes that have been written according to some design pattern. Second, design patterns are often conceptually complex code compared to traditional object-oriented code. They use higher order features such as dynamic binding extensively. If a metric gives bad values for this kind of classes, it is hard to apply it to a project, where design patterns are used. Table 3.3 shows that LCIC values for most inspected design patterns are low. Below we study these cases in detail.

For each related design pattern we consider the following questions:

1. What are the effects of applying a design pattern to the value of LCIC for all involved classes?
2. Are these effects significant?
3. Are the cases where this design pattern should be used, the same as the cases where LCIC becomes lower when the design pattern is applied?
4. How do internal cohesion metrics such as LCOM behave with respect to the design patterns?

The design patterns that are considered here are Abstract Factory, Mediator, State, Composite, Adapter, Proxy, Observer, MVC, Visitor and an anti-pattern God Class.

3.6.1 Abstract factory pattern

The Abstract Factory [44] is an example of creational patterns. This pattern can be used, when a family of classes needs to be varied at run time. In this design pattern, there are several factory classes, each with several methods as

Design Pattern	*LCIC*
Abstract Factory	High or none
Mediator	Improved
State	Minimal
Composite	Minimal
Adapter	Low
Proxy	Minimal
Observer	Varies
MVC	Low
Visitor	Varies
God Class	Varies

Table 3.3: Summary of design patterns, and their relation to LCIC.

members. These methods are simple methods that just create new objects (of certain type, related to the factory). We call these methods *factory methods*. A factory class implements an interface, which has signatures for the factory methods. Objects of factory classes are passed to other objects or are stored in a global variable. These other objects then call the factory methods to create the objects they require. Figure 3.2 shows a simple example of a class designed according to the abstract factory pattern.

LCIC is usually minimal for the factory classes, because they seldom have instance variables. If these classes have configuration variables (like `color` in the example), then either the methods use the same configuration variables, or there are different configuration variables for each factory method. The clients do not necessarily create all kinds of objects, so in the latter case, LCIC can become high. LCIC is still lower than the internal cohesion for the factory class, because there are typically no internal connections. Also, it can be assumed that a group of classes is more coherent, if the clients create all classes from the group.

On the other hand, LCIC might become higher for classes that are being created by the factory methods. If the factory method only calls the constructor, the access is ignored, because constructors are not measured in our definition of the Uses relation. If the factory method changes some attributes, the factory class becomes a client of the created class, which then only uses a small part of the client. If a class has otherwise few clients, the effect of factory methods can be significant. Compared to a case, where the abstract factory pattern has not been applied, the creating class most likely uses other features besides those needed strictly in the creation phase of the class. This suggests that LCIC becomes higher, when this design pattern is applied.

```
public interface AbstractFactory {
  I1 makeI1();
  I2 makeI2();
  I3 makeI3();
}

public class ConcreteFactoryX
implements AbstractFactory {
  private Color color;
  public I1 makeI1() { return new C1(); }
  public I2 makeI2() { return new C2(color); }
  public I3 makeI3() {
    C3 elem = new C3();
    elem.setColor(color);
    return elem;
  }
}

public class ConcreteFactoryY
implements AbstractFactory {
  ...
}
```

Figure 3.2: Abstract factory design pattern.

In practice, a class c using an abstract factory uses the created objects via interfaces (`I1`, `I2` and `I3` in the example). This can lead the LCIC computation method to make c to be a client of all kinds of object types created by *any* factory. Points-to-analysis would be needed to get rid of these non-real clients. However, this issue is not specific to the abstract factory design pattern.

It is observed that constructors are problematic to LCIC computations. If constructor calls were included in the LCIC computation, the factory methods would not have a negative effect on LCIC. The reason for ignoring the constructor calls is that usually they are done by the clients that want to use the constructed objects. If these were not ignored, these clients would use the measured class fully, and therefore they would not contribute anything useful to the LCIC measurement. The LCIC calculation could be improved, if there were a way to separate two different kinds of object constructions. In Section 3.8 one way to recognize the problematic methods is presented.

3.6.2 Mediator pattern

The Mediator design pattern [44] is a behavioral pattern that can be applied when a set of classes has complex internal dependencies. Instead of calling other objects directly to change their state, the mediator object is called instead. The mediator then calls the actual objects to change their state. This improves cohesion and decreases coupling.

The Mediator pattern can often be used to make LCIC lower. Assume there are classes `A`, `B`, `C`, `D` and `E` that all have three variables. If each class uses one variable from all of the other classes, LCIC of the classes would be 0.67. Now, if a mediator class `M` is created, all classes would have minimal LCIC, assuming that the different clients together use all variables. In general, LCIC of a mediator might be high, but LCIC values of other classes become lower.

If all involved classes have low LCIC, applying the mediator pattern does not make the LCIC values any lower. However, also in this case, the application of the mediator design pattern removes direct class relationships. In fact, this might be the most useful case to apply the mediator design pattern, because there is a maximal number of recursively coupled classes. Because of this, LCIC is not ideal for detecting, when the mediator design pattern should be applied. A more specific metric should be designed to detect the situations where the mediator design pattern should be applied. One alternative is defined in Section 3.8.

The message dispatcher architecture can be seen as an extreme case of the mediator pattern. In this kind of architecture, the components do not directly call other components. Instead, they construct messages that are sent using a message dispatcher. LCIC of the components in this kind of architecture should be low, because all the functionality of the component should be accessible using messages. LCIC of the message classes is not necessarily high, because some components can ignore parts of the messages.

3.6.3 State pattern

The State design pattern [44] is a behavioral pattern, which is typically implemented using several small classes. In this pattern, an object might be in several different states. These states are represented by state objects of different state classes. Each state object has a method, which handles an event, and returns a new state. Therefore, the state design pattern implements a kind of state machine.

Because the state objects have basically only one client, this client should use all properties of these objects. On the other hand, this kind of small class can use just few properties of their clients, or modify a single property of an object. This might make LCIC of the client classes higher.

The state objects can be understood as parts of the class, the state of which they represent. A class that describes the state is strongly connected to its client, so the internal cohesion of these classes might not be a meaningful concept.

Each state object is not intended to implement a concept, but instead is a part of a larger concept. This can be seen from the fact that the state design pattern is best modelled as a state machine, not as a class diagram.

3.6.4 Composite pattern

In the composite pattern [44], there is an interface for components, and a container class which implements the interface by a combination of objects that implement this interface. This kind of classes are often used for implementing user interface components or hierarchical document data types.

The core functionality of the classes in this pattern is implemented as one or more methods. These methods are defined in the composite class to call the same method from all contained objects. Because of this, the composite class should use other classes in the hierarchy fully via the component interface and LCIC should be low. If there is additional functionality, like for example a name for each user interface component, the container object does not access it. There should not exist very much such data.

If the component classes have a specific functionality that the component interface does not show, the clients use those component classes via different interfaces.

3.6.5 Adapter pattern

The Adapter pattern [44] can be used to change the interface of a class without changing its source code. The adapter pattern is usually needed, when adding an externally developed class to the code base. An object of this class is wrapped by an adapter object, which includes a reference to the underlying object, and methods that follow the interface are implemented by calling the corresponding methods from the wrapped class.

This kind of definition of interfaces is not supported by LCIC. If the adapter defines a limited interface, it makes the LCIC values higher. It might be natural to define this limited interface explicitly. The clients of the adapter class use it fully because it has only one variable, which represents the underlying object.

3.6.6 Proxy pattern

A Proxy object [46] can also be used to wrap an object. In this pattern, the proxy class implements the same interface as the wrapped class. Therefore, the proxy class should use all features of the class that is being wrapped.

Also the clients use the proxy class fully, because there is typically only one instance variable. Thus, the proxy class is likely to have a very low LCIC and the LCIC value of the wrapped class is mostly determined by other clients than the proxy class.

In principle, the definition of LCIC has a problem with proxy-like classes. If clients only use a few features of an actual wrapped class, then that class should have high LCIC. However, in practice it can have a low LCIC, if the actual clients are no longer direct clients of the wrapped class. To solve this problem, one could ignore classes with only one variable, and consider indirect usages via them also as direct usages. Another possibility would be to include all fields transitively, but then the use relation should be calculated transitively, too. We consider these variations in Section 3.8.

In practice the LCIC is calculated so that the clients of the proxy class are also marked as clients of the original wrapped class, because the actual clients are often made to use the proxy via an interface (that is also implemented by the actual wrapped class). If points-to analysis would be used in the LCIC calculation, the wrapped class might lose some of the non-actual clients and thus the LCIC of the wrapped class would wrongly improve.

3.6.7 Observer pattern

The Observer pattern [44] has two parts. One is the subject, which is a class to be monitored. Another is the observer, which receives events (notifications of state change) from the subject. Observers are registered into subjects, and each subject can have several registered observers. Clearly, the subject classes are clients of the observers. Observers do not have to be clients of the subject classes, because the only requirement is that they receive messages from clients. Observers are typically small utility classes that have this one singular purpose.

The Observer pattern can be implemented in two ways. If the observed subject passes itself to the observer when announcing of an event, the observer class becomes a client for the subject class and thus LCIC of the subject class has a tendency to become higher. If only the part that was changed is passed, LCIC is not affected. In our opinion the latter approach is more common. In both cases, the subject class uses the observer classes only by passing them events, and if the observer classes have other kinds of functionality and clients, the subject class might be a source of their high LCIC.

Notice that when a class is made to be observable, the clients which do not add observers will make LCIC higher. This happens because they are not using the data needed for storing references to observers. Actually the Observer pattern makes a subject class to implement another concept besides its core functionality, and in that sense this pattern is not in line

with the goals of the LCIC metric. Perhaps, one should use aspect oriented programming to separate unrelated functionality.

3.6.8 Model-View-Controller design pattern

The Model-View-Controller (MVC) pattern [44] is used to split the implementation of user interfaces into three parts called the *model*, the *view* and the *controller*. There are several ways to apply the MVC pattern. The model describes the data that can be modified via the user interface. It should not depend on the user interface at all. When the model is changed, it can signal the controller using the observer pattern, or the controller might be responsible for making all the updates into the model and the view. The view implements the outlook of the user interface. It can read the state of the model to render the outlook properly. In the Swing library, (almost all) GUI components are controllers having separate objects for the model and for the view. The controller handles the events from the view and model. For example, the model has event handlers to call the controller and the controller updates the view properly.

The LCIC of the model should be low, because the view should be able to access all attributes of the model, and the controller should be able to change all attributes in the model. At least in this case, using fields instead of methods as features is a good choice, because the view might use the getter methods of the model, and the controller might use the setter methods.

The LCIC of the view should also be low, because there should be none or just a few clients besides the controller, and the controller uses extensively the properties of the view. The LCIC of the controller is not as clear. The view and model (or rather certain event handlers used to call the controller) might be quite bad clients of the controller.

3.6.9 Visitor pattern

The Visitor pattern [44] is an example of higher order encoding of data structures. It allows adding new functionality (a visitor) to data without modifying the classes representing the data. A visitor is essentially a set of functions or algorithms applicable to a set of data types. The data in a structure are described by several classes, where each class represents an alternative in the polymorphic data structure. The classes have one method, often called `accept`, which accepts the visitor. This visitor has a method (`visit`) for each alternative of the actual data. Each class has a method that calls the corresponding method from the visitor. If the polymorphic data structure has additional data associated with the alternatives, also these are passed to the visitor.

The LCIC value of a visitor class is not an issue, since such classes often

have no instance variables. The visitors on the other hand often use directly the visited classes, and if that usage is not broad, the LCIC of the visited classes may suffer.

3.6.10 God class pattern

The God class [19] is an anti-pattern with a large class that implements functionality that should be moved to smaller classes. This kind of classes should have high LCIC, if they have several clients.

A god class might make LCIC of other classes higher. For example, it can have features narrowly complementing the behaviour of some other class (those features should be moved to that class). On the other hand, a god class can also have an opposite effect. A god class can have several narrow methods, which together use a target class broadly. If a god class is split, it should be split so that the parts using some specific class are kept together.

3.6.11 Conclusions

From the investigations above, we can conclude that LCIC is usually low for classes that are parts of design patterns. However, using design patterns might result in a design with several additional small classes. In general, it is likely that small classes make the LCIC higher.

3.7 LCIC and refactoring

The purpose of refactorings is to improve the quality of code. To evaluate, how LCIC should be used, it is important to study, how LCIC values change, when the code is modified using refactorings. It is reasonable to expect that correctly applied refactorings make the metric values smaller. Below, we consider the effect of some common refactorings on LCIC. In addition to the issues considered with design patterns, the following issues are studied.

1. When would LCIC be useful for detecting the need for a certain refactoring?
2. How can the place of a refactoring be located?
3. What happens if refactorings are performed excessively? What metric can be used to prevent this?

Metrics can help locating the classes or other modules of a program that need some refactoring. One can expect that classes with bad metric values contain design problems. When such a design problem is found by manually inspecting or by applying some automated tool, one can apply a refactoring to it.

It is however possible to go one step further. Consider a program supporting a set of automatic or otherwise specified refactorings. For each refactoring, we hope to give a meaningful metric value, which tells how desirable the refactoring is. Now, the metrics tool can list the best refactorings it finds, and the user can select the ones that seem reasonable.

The simplest way to achieve the above system is to just apply all possible refactorings, and then calculate the metrics values for the refactored programs. The refactorings with the largest improvement can then be selected. In practice, this approach is too inefficient. An efficient algorithm calculates the metric values directly for the refactorings based only on the original version of the program (possibly by estimating the metric value for the refactored program). In addition, the algorithm only needs to return the best refactorings.

More formally, we can describe a refactoring r as a mapping from programs to programs: $r : \mathfrak{P} \to \mathfrak{P}$, where $\mathfrak{P}$ is the set of programs. The refactoring metric would be $|r| : \mathfrak{P} \to \mathbb{R}$. If the value of refactoring is positive, it is a good refactoring, and if the value is negative, it is a bad refactoring. If $r(\mathfrak{p}) = \mathfrak{p}$, then $|r|(\mathfrak{p}) = 0$. A useful property for refactoring metrics could be that if $r_2(r_1(\mathfrak{p})) = r(\mathfrak{p})$, then $|r|(\mathfrak{p}) \geq |r_1|(\mathfrak{p}) + |r_2|(r_1(\mathfrak{p}))$. This means that if a refactoring is a composition of two refactorings, this composition gets as good values as its parts separately. This property means that there can be added new improved metrics for derived refactorings, i.e. if it can be detected that a combination of refactorings is good without performing it. It also implies that if a refactoring is first applied, then cancelled, it cannot have been useful. Of course, it would be even better if $|r|(\mathfrak{p}) = |r_1|(\mathfrak{p}) + |r_2|(r_1(\mathfrak{p}))$.

In the following, the refactorings Extract Interface, Extract Class, Extract Superclass, Move Field, Move Method, Call Chains and Encapsulate are considered. Table 3.4 shows the summary of effects of these refactorings on LCIC.

3.7.1 Extract Interface

In the Extract Interface refactoring [43], a new interface is created for a class.

The Extract Interface refactoring always makes LCIC lower after suitable clients have been changed to use the new interface instead. On the other hand, a class should not have too many interfaces. This is related to the depth of inheritance hierarchy, because each super-class defines an interface.

We could also have a metric that tests how many interfaces a class actually can have in the program. If a class has many clients, it can have several interfaces. If a class has only one client, then it should not have additional interfaces.

Refactoring	Bad	Good
Extract Interface	Lower	Lower
Extract Class	Higher	Lower
Extract Superclass	None	Lower
Move Field	Higher	Lower
Move Method	Higher	Lower
Call Chains	-	Lower or none
Encapsulate	-	None

Table 3.4: Refactorings and their effects to LCIC. The second column gives the effects of bad refactorings, and the third column describes the effects of good refactorings on LCIC.

If each client has its own interface to the class, then the LCIC value is necessarily the lowest possible. In Section 3.8, we investigate a variation of LCIC, where interfaces have no effect on LCIC values.

3.7.2 Extract Class

In the Extract Class refactoring [43], a new class is created from the members of the class to be refactored.

This refactoring changes LCIC in two ways. First, the clients of the refactored class might now have different LCIC. Second, the LCIC between result classes and the classes it uses can change.

The Extract Class refactoring can be used to make LCIC lower in several ways.

- If rarely used variables are grouped into one class, the created class possibly has a high LCIC value, but the LCIC value of the original class decreases.

- If the original class implements two separate features, the LCIC value of the resulting classes will decrease.

If a class is split in a way that the resulting classes have methods using different parts of another class C, this makes LCIC of class C higher. This shows that using the Extract Class refactoring might reveal further problems in the design of the program.

Detecting the precise location of refactoring can be done in two ways: find rarely used variables, or find two sets of variables, which have mostly different clients. To do the latter, variables can be given distances based on the client classes using them. The class can then be split at the longest edge in the minimum spanning tree of the variables.

3.7.3 Extract Superclass

In the Extract Superclass refactoring [43], a part of the functionality of the class is moved into a new superclass.

This refactoring makes LCIC lower, if some of the clients for the refactored class need only those features that have been moved to the superclass. Otherwise, this refactoring is useless.

Detecting this refactoring can be best done with the procedure from KABA [89]. In this procedure, points-to analysis is used to find out, what is accessed from the creation site. A new class is created for each creation site.

Also the uses-relation can be used for detection of this refactoring case. If the class has most clients using a set of features, these features are the candidate features for moving to the superclass.

Extracting an abstract superclass also creates a new interface, and therefore it makes LCIC lower. Abstract classes can otherwise be ignored, because they cannot be assumed to be complete classes, and their features are included in their subclasses anyway.

3.7.4 Merge Classes

This refactoring is the reversal of extract class refactoring. If two unrelated classes are merged into one, the LCIC value of the combined class becomes higher. If two classes that share clients are merged, LCIC of the resulting class is similar to the LCIC of the original classes.

3.7.5 Move Field

In Move Field refactoring [43], a field is moved into another class.

The Move Field refactoring should be used so that LCIC improves. This happens when a field is moved from a class where it was not used with other fields, to a class where it is used together with other fields. Methods that include internal accesses to the field should also be moved.

If a field is moved to a class that includes unrelated fields, LCIC becomes higher, because the classes that require this field will now get unrelated fields. If the original fields were unrelated too, the cohesion might not change. This kind of fields also cause low internal cohesion.

The effect of this refactoring for a class with n fields is at most $1/n$. If several fields are moved simultaneously, the effect might be respectively larger.

Candidates for this refactoring can be detected using the Uses relation as follows. First, find a group of fields that are often used together, but other fields are not used. These fields are the candidates for fields to be moved. Second, find a class which is used by the clients that only use this

group of fields. This class is a candidate for the class where the fields should be moved into. If such a class is not found, a new class can be created.

This refactoring can be done in two steps: First extract a class with a group of fields, then merge this with the another class. On the other hand, extracting a class could be done in two steps: create an empty class, and then move members to this class.

3.7.6 Move Method

In Move Method refactoring [43], a method is moved into another class.

Because the fields do not change, the LCIC of the classes directly involved in the refactoring does not change.

If the method caused the original class to be a client for a class A, and the method uses only a few features from that class, LCIC of class A becomes lower, if the target class is already a client of this class. Also the reverse might be true: The added method might be the only one using a client class. In this sense, LCIC follows the good uses of the refactoring.

If the method accesses internal fields of the class it was originally in, the new class becomes a client for the original one. This might make the LCIC of the original class lower or higher, depending on how much the new class already used from the original one.

If the method accesses internal fields of the class it was moved into, the LCIC between the original and new classes might become higher.

The effect of direct usage is as follows: The classes using this method no longer need to be clients for the original class, but they are now clients of the new one. If the method accesses some internal fields of the new class, the LCIC of the new class becomes lower. If the method accesses some fields of the old class, the LCIC of old class becomes higher.

Based on the effects above, a candidate for moving a method can be found as follows. First, the candidate method uses the class, into which it should be moved, more than the class in which it is. Second, the clients that use this method are mostly the clients that use the new class, instead of being clients of the old one. Third, the set of used classes for the method intersects with the set of used classes in the new class.

3.7.7 Move Field in Inheritance Hierarchy

Moving Field in Inheritance Hierarchy [43] is an important special case of the Move Field refactoring.

For example, if a field that is not used in the superclass is moved down in the inheritance hierarchy, LCIC of the superclass improves. LCIC of the subclass stays the same, because in both cases the class includes the field.

Searching for candidates for this refactoring can be done in the following way: If there is a field that is only used by one subclass of the superclass, it can be moved into that subclass.

A field that should be moved into a superclass, cannot be found that easily. If there were such a field, then also other subclasses should have this kind of field. Unfortunately, recognizing similar parts from classes is hard to do automatically.

3.7.8 Encapsulate Fields

In the Encapsulate Fields refactoring [43], public fields are changed to accessor methods.

The Encapsulate Fields refactoring has no effect on LCIC. This is in contrast to LCOM, which becomes higher, if an accessor method is added. Some definitions of LCOM do not take indirect access into account, so replacing internal accesses by the new accessor methods makes the cohesion even worse.

3.7.9 Eliminate Call Chains

In Eliminate Call Chains refactoring [43], accesses such as `a.b().c()` are changed into accesses `a.bc()` and a new method `bc` is created.

The need for the Eliminate Call Chains refactoring can be sometimes detected using LCIC. The clients have a reference to an object, and they use the call chain to determine some property of a subobject of this object. When the call chain is eliminated, the client will not anymore be a client for the class of the subobject.

One can also go overboard with this refactoring, and expose all methods of subobjects by adding delegate-methods to the containing object. The number of methods metric is one possibility for detecting this. Another metric would be the ratio of simple methods and number of fields.

If several fields are accessed using chaining on the same field, it is not necessary to refactor the class, and in this case LCIC does not give high values.

Another refactoring related to call chains is the Create Temporary Variable refactoring. This refactoring cannot be found using LCIC. Star diagrams [72] can be used to find this refactoring.

3.7.10 Eliminate Conditionals

One of the more complex refactorings is changing conditionals to dynamic binding. This refactoring generates several small classes. These might use only a small part of the classes used by the original large class, making LCIC higher. This kind of classes are closely related to the class they are using.

For example, a class might represent a test for the used class to determine if a field has a certain value. This kind of classes should be collected into a class, where they form inner classes, and they should together define one client, which uses all features of the class they are related to.

3.8 Variations

Analysis of the experimental results (Section 3.5) and the theoretical validation using design patterns and refactorings (Sections 3.6 and 3.7) revealed several possible improvements to LCIC.

One type of classes that have high LCIC values are classes that contain a lot of data and have clients that modify or use a small part of this data. This suggests that classes might not be the right solution as clients of the classes.

Also, for classes that have only one field, it is hard to get useful information about cohesion. A possible solution would be to check, how the clients of this class use the underlying field. Several design patterns create this kind of classes, at least in the most pure form of these patterns.

It was also found out that adding interfaces might make LCIC artificially low. Because of this, we consider a variation where the interfaces have no effect on results.

It turned out that in some cases the direct calculation of client relationships does not give good results, so we define two variations for calculating client relationships: transitive calculation, and calculation that handles single variable classes in a special way.

We also propose methods to detect factory-like methods and possible sets of classes to apply the mediator pattern.

3.8.1 Ignoring interfaces

It is possible that interfaces for classes have been badly designed. In this case, it might be useful to ignore interfaces when calculating LCIC. We give a version of LCIC that ignores interfaces:

$$LCIC_{ii}(\mathfrak{c}, \mathfrak{d}) = 1 - \frac{|\{\mathfrak{f} \in \mathfrak{d} \mid \text{Uses}(\mathfrak{c}, \mathfrak{f})\}|}{|\mathfrak{d} \cap \mathfrak{F}|}$$

Instead of the fields accessible by the client, all fields in the class are included.

The disadvantage of using this simplified version of LCIC is that in most cases the interfaces are well designed, which causes false positives.

Of 1201 classes with LCIC values in jEdit and related part in JDK, 615 classes get higher values when interfaces are not used. There are following reasons for this:

- There are private fields, which are never used by the methods of the class. Because of this, they are not accessible from any external interface. It is also possible that some variables become impossible to use by inheritance.

- The class is possibly used through a very generic interface, for example it implements a `toString`-method. The effect of this kind of interfaces becomes too big, if large programs are used. There are at least two ways to avoid this. The first is to give less weight to this kind of accesses, based on how many classes implement the interface. The second one is to use points-to analysis to rule out impossible clients.

  The simplest solution is to omit `java.lang.Object` from the calculations. After this has been done, 506 classes with higher values remain. In only 296 cases the difference between LCIC and $LCIC_{ii}$ is larger than 0.1.

- In the remaining cases, the clients use the classes via limited interfaces. There are several reasons why not everything is accessible via the interface.

One way to give different weights for client c of class s based on the interfaces is

$$\text{weight}(\mathfrak{c}, \mathfrak{d}) = \frac{1}{\min\{\text{classes}(\mathfrak{v}) \mid \mathfrak{v} \subseteq \mathfrak{d}, \mathfrak{f} \in \mathfrak{d}, \text{Uses}(\mathfrak{c}, \mathfrak{f}, \mathfrak{v})\}}$$

where classes($\mathfrak{v}$) is the number of classes implementing the view that corresponds to interface $\mathfrak{v}$. This weight is smaller for classes that access the other class via a common interface.

Table 3.5 shows the average LCIC values for classes with a different number of interfaces. It is observed that there is strong positive correlation between the number of interfaces and LCIC. This is probably because the classes with a lot of interfaces are generally deep in the inheritance hierarchy. On the other hand, there is some correlation between the number of interfaces and the difference of $LCIC_{ii}$ and LCIC values.

We also measured, how well the interfaces cover the classes that implement them. Usually the interfaces cover almost everything from the classes. In some cases, there are limited interfaces that only access very small parts of the class implementing them.

3.8.2 Transitive call relation

One possible variation for LCIC is to use a transitive call relation instead of the direct one. In this case, classes will be clients of the measured class, if some of the classes they use, use transitively the measured class.

Interfaces	Classes	Average LCIC
2	204	0.30
3	309	0.21
4	232	0.21
5	198	0.25
6	78	0.32
7	58	0.38
8	30	0.44
9	63	0.48
10	37	0.54
11-	188	0.57

Table 3.5: Numbers of classes with a certain number of interfaces. Only the class `Object` has one interface. The largest amount of interfaces is 39.

The transitive call relation is defined as follows:

$$\text{Uses}^*(\mathfrak{c}, \mathfrak{f}, \mathfrak{v}) = \text{valid}((\mathfrak{c} \cdot \Sigma^*) \cap (\Sigma^* \cdot (\Sigma - \max(\mathfrak{f})) \cdot \text{internal}_{\mathfrak{v}}(\mathfrak{f})))$$

The relation $\text{Uses}^*(\mathfrak{c}, \mathfrak{f}, \mathfrak{v})$ holds, if class $\mathfrak{c}$ has a method $\mathfrak{m}$ which transitively calls via view $\mathfrak{v}$ a method that accesses field $\mathfrak{f}$.

The problem with using this definition for LCIC is that intuitively the direct relationship has much more meaning than the transitive one.

When considering transitive calls relations, it becomes harder to define what is meant by the client interface. We define it to mean the union of interfaces used in all transitive calls from the client class.

The transitive call relation can be used in another way. If a client directly uses a variable, also transitive uses can be included:

$$\begin{aligned}
\text{View}^*(\mathfrak{c}, \mathfrak{d}) &= \bigcup_{\mathfrak{f} \in \mathfrak{d}, \mathfrak{v} \subseteq \mathfrak{d}, \text{Uses}^*(\mathfrak{c}, \mathfrak{f}, \mathfrak{v})} \mathfrak{v} \\
\text{accessible}^*_{\mathfrak{c}}(\mathfrak{d}) &= \{\mathfrak{f} \in \mathfrak{d} \mid \text{Uses}(\text{View}^*(\mathfrak{c}, \mathfrak{d}), \mathfrak{d}, \mathfrak{f})\} \\
LCIC^*(c, \mathfrak{d}) &= 1 - \frac{|\{\mathfrak{f} \in \mathfrak{d} | \text{Uses}^*(\mathfrak{c}, \mathfrak{f})\}|}{|\text{accessible}^*{}_{\mathfrak{c}}(\mathfrak{d})|} \\
LCIC^*(\mathfrak{d}) &= \frac{\sum_{\mathfrak{c} \in \text{clients}(\mathfrak{d})} LCIC(\mathfrak{c}, \mathfrak{d})}{|\text{clients}(\mathfrak{d})|}
\end{aligned}$$

The cases where transitive calculation gives better results could be cases where a piece of data is first passed to a second method which accesses it, and then passed to a third method, which accesses it more. However, this

does not seem to be good design. It would be better if the accesses to data were in one method, because this would improve the cohesion.

Another way to use the transitive call relation is to have a transitive member relation, too. Instead of fields of the class, one can also inspect all fields of the contained objects. In this case, it could be useful to consider the call relation transitively.

If types($\mathfrak{f}$) is the set of views of possible classes that field $\mathfrak{f}$ can point to, the set of all possible fields accessible from class $\mathfrak{c}$ can be defined as:

$$\begin{aligned} \text{fields}(\mathfrak{c}) &= \bigcup_{\mathfrak{f} \in \mathfrak{c}} \text{types}(\mathfrak{f}) \cup \bigcup_{\mathfrak{c}' \in \text{types}(\mathfrak{f})} \text{fields}(\mathfrak{c}) \\ \text{fields}_{\mathfrak{v}}(\mathfrak{c}) &= \text{fields}(\text{accessible}^*_{\mathfrak{v}}(\mathfrak{c})) \end{aligned}$$

Now, the transitive version of LCIC can be defined as:

$$LCIC^{tr}(\mathfrak{c}, \mathfrak{d}) = 1 - \frac{|\{\mathfrak{f} \in \text{fields}(\mathfrak{d}) \mid \text{Uses}^*(\mathfrak{c}, \mathfrak{f})\}|}{|\text{fields}_{\mathfrak{c}}(\mathfrak{d})|}$$

It is not clear how to take interfaces into account in this variation.

3.8.3 Ignoring classes with only one variable

Classes that have only one variable do not contribute anything useful to normal LCIC calculation. These classes could be handled so that if a class is used via them, it is considered to be a direct usage.

This variation can be easily defined. First we define sets of classes with different numbers of variables $\mathfrak{C}_n = \{\mathfrak{c} \in \mathfrak{C} \mid |\mathfrak{c}| = n\}$. Then we can define possible paths for class $\mathfrak{c}$ accessing field $\mathfrak{f}$ via interface $\mathfrak{v}$ as $\mathfrak{c} \cdot (\mathfrak{C}_0 \cup \mathfrak{C}_1)^* \cdot \text{internal}_{\mathfrak{v}}(\mathfrak{f})$.

The classes in jEdit use variables from 1201 classes. Of these, 142 have only one variable. Some of these classes need only one variable to implement some simple functionality. Some are abstract classes, where this variable is not related to the central functionality of the class. Quite small part of these classes implemented adapter or some other design pattern.

3.8.4 Interpretation of clients

It was found out that the notion of clients is the most difficult to define properly. One simple variation is to consider only declared methods in the class instead of all inherited methods. Experiments were made to find out on what kind of classes this makes a difference.

There are two interpretations for what does a method access. One can consider just the direct accesses, or all indirect accesses can be taken into

account, also the ones via the parent class. In the experiments, indirect accessed were included.

In theory, the non-flat clients have some advantages. They are usually more cohesive than the flat versions. They also have less classes they use, because they are smaller. However, our experiments show that only 25 classes of 1187 get more than 0.1 decrement in LCIC when non-flat clients are used, while 326 classes have more than 0.1 increment. The decrements only happen, when there is a parent class that uses some class partially. The increments happen when there are child classes that use some class partially. The latter case seems to be more common.

There are other possibilities that could be used as interpretations of clients, like:

- Packages are seen as clients of classes. In this case, the classes with high LCIC can be assumed to have problems.
- In addition to inherited classes also the child classes can be included. In this case, inheritance hierarchies are seen as clients.
- Classes are classified manually to define the clients.

Exploring these alternatives is left as a topic of future work.

3.8.5 Reverse LCIC

No matter how cohesive the class is, the clients can still use it differently from what was intended. To analyze this phenomenon metric values can be calculated from the inverse of the use-relation. Denote by rLCIC the external cohesion metric calculated for the inverse use-relation. A class gets high rLCIC if it does not use all features from the classes that it uses.

rLCIC is calculated as follows:

$$rLCIC(\mathfrak{c}) = \frac{\sum_{\mathfrak{d} \in \mathfrak{C}, \mathrm{Uses}(\mathfrak{c},\mathfrak{d})} LCIC(\mathfrak{c}, \mathfrak{d})}{|\{\mathfrak{d} \in \mathfrak{C} \mid \mathrm{Uses}(\mathfrak{c}, \mathfrak{d})\}|}$$

If there is a class that has only one feature that the client uses, this immediately makes the rLCIC high. Some classes, like `java.lang.Class` are like this. Because `java.lang.Object` uses `java.lang.Class`, if flattened clients are used, all classes get lower rLCIC.

After the effects of `java.lang.Object` have been eliminated, rLCIC still gives generally higher values than LCIC. The classes that get high cohesion, usually use a set of many classes through some interface. This use can happen in the parent classes. In its current form, rLCIC does not produce very interesting results.

3.8.6 Detecting creation methods

There are two uses for created objects: either they are used by the creator, or they are returned from factory-like methods, which we call here *creation methods*. These creation methods should not use the objects they create. There are several uses for being able to detect this kind of methods. One application is to improve the LCIC metric. There, the factory methods could be left out from the computation, because they are not real clients for the classes they create. Another application would be in points-to analysis. There we could use creation method calls as creation sites.

Our procedure for detecting creation methods is the following: Assume that a creation method creates an object of a certain class. Then, intraprocedural analysis is used to determine, if the created object can be returned from the object. Passing the created object to a field or another method is ignored.

Of the 19963 methods in the test group (jEdit and JDK parts), 3856 methods create objects. Of these methods 792 (around 4% of all methods) were detected as creation methods. These were of following kinds:

- Methods that construct objects from arguments or global data by for example parsing.
- Methods for copying a target object.
- Accessor methods that return temporary objects for accessing the class further.
- Accessor methods implementing singleton design pattern.
- Methods that implement functions on immutable objects.
- Actual factory methods, which just create new objects.

Almost all detected methods only created the objects, and did not use them to implement their functionality. A large amount of objects that were created by undetected methods were exceptions and utility objects like `StringBuffer`-objects.

3.8.7 Detecting candidates for mediator pattern

When designing a metric to detect candidates for applying the mediator pattern, one has to find cliques from the usage graph of the program. It can be assumed that the pattern should only be applied, when there are recursive relationships involved. In the case of non-recursive relationships, the classes should just be divided into layers. Candidates for applying the mediator pattern can be found in the following way. First, construct a symmetric

relation between classes that use each other. This relation relates classes that have a direct recursive relationship. Then, inspect classes which have many such related classes.

We can define the relations for use with different orders, where use of order 0 is a direct use, use of order 1 is a use via one other class and so on. If all recursive use relationships are considered, it might give unrealistic results, at least if no points-to analysis is made (for example container classes would have this kind of relationships). Let I_n denote the set of paths where there are n interclass calls. More formally define that the call path w is in I_n, if the path contains n possibly overlapping occurrences from set $\bigcup\{\mathfrak{c}\mathfrak{c}' \mid \mathfrak{c}, \mathfrak{c}' \in \mathfrak{C}, \mathfrak{c} \neq \mathfrak{c}'\}$, which is the set of interclass calls. Further, denote the set of paths where there are n or less interclass calls by $I_{\leq n} = \bigcup_{m \leq n} I_m$. Now we can define use relation with order n as

$$\text{Uses}^n(\mathfrak{a}, \mathfrak{b}) \quad = \quad \text{valid}(\mathfrak{a} \cdot \Sigma^* \cdot \mathfrak{b} \cap I_{\leq n})$$

A relation and a metric to search for candidates for applying the mediator design pattern can now be defined as

$$M_n(\mathfrak{a}, \mathfrak{b}) \quad = \quad \text{Uses}^n(\mathfrak{a}, \mathfrak{b}) \wedge \text{Uses}^n(\mathfrak{b}, \mathfrak{a})$$

$$\mu_n(\mathfrak{a}) \quad = \quad |\{\mathfrak{b} \in \mathfrak{C} \mid M_n(\mathfrak{a}, \mathfrak{b})\}|$$

Good values for n should be quite small (0,1 or 2), because even if the cliques are bigger, at least a part of them should be found already with small values.

3.8.8 Points-to analysis

Points-to analysis can be used to make resolving method calls more precise. This means that we get a new call relation Call_{pt}. This relation is constructed similarly to the old one, but now we also have available the results from points-to analysis, which allows us to reduce alternatives when resolving dynamic binding.

There are two advantages in using a points-to analysis. First, points-to analysis makes the analysis more efficient. Second, points-to analysis also reduces the number of artificial clients in the analysis. The latter could also be a disadvantage, because reducing the amount of possible clients might make the results of LCIC dependent on a small set of clients.

Initial experiments show that the number of virtual method calls can be reduced to 1/10'th of what was the naive approach.

Some challenges remain in defining a suitable points-to analysis for LCIC. For example, if objects are stored into containers such as `ArrayList`, when an element is read from the container, a simple points-to analysis would assume that it could be of any type that has been stored in that kind of container.

3.9 Conclusions and Further Work

We proposed a metric LCIC that measures how coherently the clients use a class according to the roles that have been specified for it. The proposed metric can be used to find several different kinds of design problems such as classes that should be split, and unnecessary accessor chains. Using these metrics, a number of design problems were localized in popular open source projects.

By examining programs with different kinds of LCIC values and reasoning about the behaviour of LCIC when using refactorings or design patterns, we derived several variations of LCIC. Some variations were shown to be improvements and some were only useful to confirm original design decisions. Some variations need more experimental studies before any conclusions can be drawn.

Finding the intended purposes of methods would be useful for metrics calculation and program comprehension. As a starting point we defined a simple analysis for finding factory methods. This analysis can be used to omit factory methods from LCIC calculations and make points-to analysis more precise.

To fully leverage the power of metrics, one can use a combination of metrics. More analysis should be conducted in order to state what kinds of problems the metric proposed in the present study can help to detect, and what kind of metric combinations can be used to find these flaws. Our intent is also to search for other client based metrics and test their applicability. As an example we are going to experiment with a metric to search for cliques of classes, where the mediator design pattern should be applied.

There are even more ways to define variations for LCIC. One possibility would be to consider methods as features instead of fields. Another possibility would be to change the calculation of LCIC so that classes with more clients are given more weight. This kind of metrics would point out the most used classes with problems. A more complex approach would be to consider client relations as slices. Another alternative is to develop metrics, which measure the lifetime of an object using some kind of flow analysis.

The relationship between metrics and refactorings must be elaborated further. An interesting approach would be to consider a set of refactorings and a metric, and then find the refactored program, which gives the minimal metric value. There are at least two challenges that must be faced to get started. First, one has to fix an example, which suits for this kind of research. Second, one needs to find out, what kind of model of programs is needed for this example. Once this has been done, one could be able to show the relationships between the selected metrics and refactorings.

Another related approach is to define metric values directly for refactorings, instead of program structures such as classes. In this case, the user can

just select the best refactorings to improve the quality of the program. In the next chapter we will give such a metric. We will also show that this kind of refactoring metric can be defined in a way that essentially encompasses intuitive ideas about modularity.

Chapter 4

Refactoring Metrics

4.1 Introduction

In this dissertation, the interest is in improving the quality of programs by the aid of software metrics. If a metric value for a software artifact indicates bad quality, a design problem can often be identified [56]. In the previous chapter, it was shown that the metric LCIC is suited for finding many such design problems. The next step is solving the design problem by *refactoring* the source code [43]. The basics for refactoring were discussed in Section 1.6.

In current development environments such as Eclipse, both metrics calculation and refactoring can be done automatically. To automate the process further, a *refactoring suggestion system* [71] can be used. Using metrics, these systems suggest refactorings that should improve the quality of software. Without a suggestion system, the refactoring has to be determined manually by inspecting the source code or a model of the program. Using the experience gained by inspecting cohesion metrics, we propose a suggestion system for the design problems that are related to cohesion and modularity in general.

The previous suggestion systems are based on the following idea: The metrics values are first calculated, and then tested against threshold values. Then a boolean expression is used to determine whether a refactoring is suggested or not. In addition to software metrics with threshold values, code smells [43] can be used. A code smell is a "warning sign" that is considered to indicate a design problem. Formally they are just binary predicates that test for some feature of a relevant part of the program.

As an alternative for this method, the use of *refactoring metrics* is introduced in the present work. These metrics are applied directly to possible refactorings. The advantage is that by using refactoring metrics, the most important suggestions can be recognized. Because refactoring metrics give

suggestions for improvements, they are also more reliable in finding design problems than traditional metrics. This is because the traditional metric values might depend on the semantic purpose of the software artifact, but refactoring metrics measure refactorings void of any semantic content. The refactoring metrics are a further development of an idea in Chapter 2, where the difference between metric values was seen as the most useful indicator.

In this study, we concentrate on Move Member -refactorings, where fields or methods of classes are moved into other classes. These refactorings are related to coupling and cohesion of classes. CBO and LCOM [26] are the original examples of metrics that measure coupling and cohesion. LCIC measures the cohesion of classes by inspecting the clients of the classes. In Chapter 3 it was observed that high LCIC values often imply design problems, some of which are related to Move Member -refactorings. These observations will be used here when proposing a refactoring metric.

Because each refactoring might have several different kinds of heuristics that can be used to determine whether a refactoring operation should be applied (see for example [12]), some care is needed when defining refactoring metrics. Basically a *cost function* is formed for the system, and then it is evaluated how the refactorings change the value of the cost function. The cost function should be selected so that the cost grows when the metric values associated with different heuristics grow.

The plan of the chapter is following: First related work is discussed. In Section 4.3 a definition is given for the proposed refactoring metric and the associated suggestion system. In Section 4.4 the properties of a selected open source program are analyzed using the cost function and traditional metrics. Section 4.5 contains an analysis of individual suggestions generated by the proposed suggestion system. In the last section we make conclusions.

4.2 Related research

Most research on refactoring has been focused on finding new kinds of refactorings and performing refactorings correctly. The research for software metrics has been concentrated on finding design problems.

Du Bois et al. [12] present metrics based guidelines for applying some refactorings. Their guidelines for Move Member refactoring are

- Move methods that do not use local resources.
- Move methods that are not called internally.
- Move methods that are used by a single external class.

These heuristics are similar to the ones discovered by the metric of the present research.

Lanza and Radunescu [56] present several metrics based detectors for design problems. Similar detectors are used for suggestion systems such as T-Rex [71] for TTCN-3 test specifications.

Bryton and Abreu present a suggestion system for modularity oriented refactoring [20]. They intend to find out which metrics would be useful by inspecting refactored software and comparing their metrics. No specific metrics or results were discussed in their work.

Sarkar et al. [86] propose a set of metrics that can be used for automatic or semi-automatic modularization of source code. Their metrics are more related to packages than classes. They propose that modules should be formed based on similarity of service. They argue that internal call relationships should not be used as an indicator of cohesion. The same approach is used in our refactoring suggestion system, where mostly external relations are considered.

Joshi and Joshi [52] observed that current metrics are too coarse-grained to be used for refactoring. They proposed microscopic coupling metrics RMC and RIC that apply to relations between methods and coupled classes. In our terminology, the metrics RMC and RIC are refactoring metrics for the Move Member refactoring. These metrics work similarly to our metric in the sense that they count the relationships between methods and classes. A similar work is the fine-grained metrics [35] proposed by English et al. They used the metrics to investigate how the C++ friend construct is used.

Simon et al. [88] present a distance based metric and visualization to assist in refactoring. This work is probably the closest to our approach. Their definition of distance between methods is similar to our notion of probability. They define distance as follows: Each method is represented by a set of used members. If sets are A and B, then their distance is

$$\frac{|A \cap B|}{|A \cup B|}$$

Our formula for determining whether two singleton modules should be combined is

$$\frac{|A \cap B|}{|A| + |B|} < \frac{o}{2}$$

The visualization based approach of Simon et al. might not scale well into large programs.

4.3 Motivation and Definition of the Metric

To define the refactoring metrics, one has to have a suitable model of the programs. Since we are only interested in the module structure of the program, it is just assumed that there is a set $\mathfrak{A}$ of *atoms* and a set $\mathfrak{U}$ of *modules*

or compilation units. The *modular structure* is a function $\mathcal{M} : \mathfrak{A} \to \mathfrak{U}$ that returns the module containing the atom. Similarly $\mathcal{M}^{-1}(\mathfrak{u})$ returns the atoms in unit $\mathfrak{u}$. It is assumed that there are k atoms i.e. $|\mathfrak{A}| = k$. In our implementation of the metric for Java, the atoms are the methods and fields of classes, and the modules are compilation units or files. The reasons that compilation units are used instead of classes are that (i) inner classes cannot be separated from their outer classes when considering modularity, and (ii) if there are several classes in a compilation unit, they are usually closely related.

4.3.1 Cost function

In the previous chapters we were only interested in the cohesion, which is just one design heuristic that is related to modularity. Other such heuristics are coupling and size of modules. It is not a good idea to define the suggestion system based on these metrics, because there is a far superior way to approach the issue. This is simply to take a step back, and forget about these heuristics. We simply calculate the *cost* of the module structure, which tells directly how good the module structure is. The design heuristics are still useful, however. They can be used to make sure that the definition of the cost function follows our intuition about good design. To develop a good cost function, it should agree with as many heuristics as possible. It is also possible that our intuitions are proven false.

To find a suitable cost function, the following approach is used: Consider a very simple programming task, and estimate how long it takes to perform this kind of task. Refactorings should make this time smaller. The task we select is understanding the implementation of a method. This kind of understanding is needed for example if a programmer wants to modify the functionality of the method, or a programmer is inspecting the sources to find a bug. It is assumed that this task is related to the modular structure of the program, because it is in general harder to understand a method if it calls methods from many different modules. Also, if the called modules are large, understanding and writing the method is hard, because it may be hard to find the called method from a large class. The method can be found in some other way, but then the module structure was not used.

To give a more precise description of the programming task, it is assumed that the programmer wants to understand the implementation of atom $\mathfrak{a}$. To do this, she needs to inspect all related atoms to understand their functionality. Because the atoms are structured into modules, the related modules need to be inspected. In the current research, the related atoms are the atoms that are called from atom $\mathfrak{a}$. In a more advanced model, also atoms that call atom $\mathfrak{a}$ could be included, because it might be useful to understand how the atom is used.

As a summary, the proposed methodology has the following advantages

- For each case, we get a concrete explanation why the system makes the suggestion.
- The system can be improved by taking more features into account in the cost function.
- User interface research can be used to assist in developing the cost function.
- The cost function does not depend on the semantics of the program.

4.3.2 Definition of cost function

To define a cost $\gamma(\mathcal{M})$ for modular structure $\mathcal{M}$, we start from simple programming tasks. Formally, the tasks that programmers perform can be thought to be events in a probability space. Each task $\mathfrak{w} \in \mathfrak{W}$ has a set of involved atoms (fields or methods) $\mathfrak{A}(\mathfrak{w})$. The probability space can now be used to describe the relations between atoms:

$$P(\mathfrak{a}) = P(\{\mathfrak{w} \in \mathfrak{W} \mid \mathfrak{a} \in \mathfrak{A}(\mathfrak{w})\})$$

Here $P(\mathfrak{a})$ is called the *probability of the atom*. This can be seen as a generalization of using graphs [15] or hypergraphs for describing relationships between atoms. The *probability of a module* $\mathfrak{u}$ depends on the probability of its atoms, i.e. $P(\mathfrak{u}) = P(\mathcal{M}^{-1}(\mathfrak{u}))$. Once these probabilities have been calculated, it can be assumed that they approximate also the relations caused by future methods.

The programming task we are now interested in is understanding the implementation of a method by finding the descriptions of the called methods. If the module structure is not used, the task is not related to the module structure. Assume that the probabilities for looking up the methods are uniform and independent.

The *cost of inspecting a module* $\mathfrak{u}$ can be defined in several ways. It is defined here simply as

$$\gamma(\mathfrak{u}) = |\mathcal{M}^{-1}(\mathfrak{u})| + o$$

Here we assume that inspecting each member of the module costs 1, and inspecting a new module additionally costs o units. The first component of $\gamma(\mathfrak{u})$ measures the *size* of the module. It can be thought that larger modules are harder to understand. Adding the term o makes the cost function measure the *coupling* of the system, because it counts how many modules are used by the atoms. Therefore we call o the *coupling factor*. The coupling factor is a constant that should be determined before evaluating the system cost. Increasing the coupling factor makes the optimal modules larger. If

the coupling factor is 0, the system considers modules with size 1 to be optimal. On the other hand, if size is not taken into account, having only one module would produce optimal results.

To estimate the average cost of a programming task, the probability that a module is needed has to be calculated. For each module $\mathfrak{u}$, the cost is the probability that the module is needed in a task $P(\mathfrak{u})$ multiplied by the cost of inspecting the module $\gamma(\mathfrak{u})$. The total *cost for module structure* $\mathcal{M}$ is then

$$\gamma(\mathcal{M}) = \sum_{\mathfrak{u} \in \mathfrak{U}} P(\mathfrak{u})\gamma(\mathfrak{u})$$

The module structure $\mathcal{M}$ is *optimal*, if there is no such $\mathcal{M}'$ that $\gamma(\mathcal{M}') < \gamma(\mathcal{M})$.

To generate suggestions for refactoring, one should be able to estimate the probability of any set of atoms. First it is necessary to approximate the probability of a single atom $\mathfrak{a}$. To accomplish this a *relation vector* of $\mathfrak{a}$ is used, notated $\overline{\mathfrak{a}}$. It is first assumed that there is a k-dimensional vector space with basis $A = \{\mathfrak{a}_1, \ldots, \mathfrak{a}_k\}$, which is equal to set of tasks. Basically, the relation vector represents the weights of the users of the atom.

$$\overline{\mathfrak{a}} = \sum_{i=1}^{k} x_i \mathfrak{a}_i$$

where $0 \leq x_i \leq 1$. Note that $\mathfrak{a}_i$ is a basis vector, similar to $\mathbf{i}$ and $\mathbf{j}$ in the usual notation $x\mathbf{i} + y\mathbf{j}$. To give relation vectors values, consider dot product $(\overline{\mathfrak{a}}, \mathfrak{b})$ with another atom $\mathfrak{b}$. In the implementation, $(\overline{\mathfrak{a}}, \mathfrak{b}) = 0$ if $\mathfrak{b}$ does not directly call or access $\mathfrak{a}$. If $\mathfrak{b}$ calls $\mathfrak{a}$, and there are n different virtual methods that could be called on the call site, then it holds that $(\overline{\mathfrak{a}}, \mathfrak{b}) = \frac{1}{n}$. Currently, only type information is used to determine, which virtual methods can be called on a call site. From the definition it follows that if there is only one alternative method that can be called, then $(\overline{\mathfrak{a}}, \mathfrak{b}) = 1$. If virtual methods are ignored, then $\overline{\mathfrak{a}}$ is simply the set of methods that call $\mathfrak{a}$.

Because each task is thought to have the same probability, it is defined that the *probability of a vector* $\mathbf{v} = \sum_i x_i \mathfrak{a}_i$ is

$$P(\mathbf{v}) = \frac{\sum_i x_i}{k}$$

The *combination* of vectors $\mathbf{v} = \sum_i x_i \mathfrak{a}_i$ and $\mathbf{v}' = \sum_i y_i \mathfrak{a}_i$ is defined as

$$\mathbf{v} \oplus \mathbf{v}' = \sum_i \max\{x_i, y_i\} \mathfrak{a}_i$$

The idea is that it is assumed that

$$P(\{\mathfrak{b}_1, \ldots, \mathfrak{b}_m\}) = P(\overline{\mathfrak{b}_1} \oplus \ldots \oplus \overline{\mathfrak{b}_m})$$

For example $\overline{\mathfrak{a}} \oplus \overline{\mathfrak{b}}$ would represent the module that has atoms $\mathfrak{a}$ and $\mathfrak{b}$. Now the relation vectors can be used to calculate the probability of any module. Notice that if virtual calls are ignored, the combination is simply a set union. Another possibility to define the combination would be

$$\mathbf{v} \oplus \mathbf{v}' = \sum_i (x_i + y_i - y_i y_i)\mathfrak{a}_i$$

Notice that both definitions of $\oplus$ are associative and commutative. This can be used to make the implementation of the suggestion system more efficient.

The above notion of probability is related to *cohesion*: If atoms have a similar set of users or related modules, they have similar associated vectors. A cohesive class contains similar atoms, and therefore it should have a low probability compared to the probability of its parts. If a class has low cohesion, the probability is close to the sum of the probabilities of its parts. Another factor that probability measures is *significance*. A significant class is used more often than other classes, which makes its probability higher.

Notice that $\gamma(\mathcal{M})$ can be given by a direct formula based on the costs of tasks instead of costs of units

$$\gamma(\mathcal{M}) = \sum_{\mathfrak{b} \in \mathfrak{A}} \sum_{\mathfrak{u} \in \mathcal{M}(\mathfrak{A}(\mathfrak{b}))} \max\{(\mathfrak{b}, \overline{\mathfrak{a}}) \mid \mathfrak{a} \in \mathcal{M}^{-1}(\mathfrak{u})\}\gamma(\mathfrak{u})$$

It is easy to see that this formula is equivalent to the definition of $\gamma(\mathcal{M})$ given before.

4.3.3 Suggestion system

We have now shown how to calculate the cost $\gamma(\mathcal{M})$ of the system. The formula for the cost takes into account the size, the significance and the cohesion of the modules, and also the coupling of the system. We consider these to be the most important design heuristics related to the modularity of programs. Next it is shown how the cost function can be used to make suggestions to improve the system.

Consider three simple refactorings that can be used to improve the modular structure of the programs:

- Detach an atom $\mathfrak{a}$ to a new module $\mathfrak{u}$: $\text{detach}_{\mathfrak{u},\mathfrak{a}}$.
- Combine two modules $\mathfrak{u}$ and $\mathfrak{u}'$: $\text{combine}_{\mathfrak{u},\mathfrak{u}'}$.
- Move an atom $\mathfrak{a}$ from module $\mathfrak{u}$ to another module $\mathfrak{u}'$: $\text{move}_{\mathfrak{u},\mathfrak{u}',\mathfrak{a}}$.

Notice that Extract Class -refactoring can be implemented as a detach and several moves. However, the extraction is then implemented as several steps, and therefore the best suggestions for extracting classes might not be found.

Let us formalize the refactoring suggestions by using the mapping $\mathcal{M}$. If $\text{combine}_{\mathfrak{u},\mathfrak{u}'}(\mathcal{M}) = \mathcal{M}'$, then

$$\begin{array}{lcll} \mathcal{M}'(\mathfrak{a}) & = & \mathfrak{u} & \text{if } \mathfrak{a} \in \mathcal{M}^{-1}(\mathfrak{u}') \\ \mathcal{M}'(\mathfrak{a}) & = & \mathcal{M}(\mathfrak{a}) & \text{otherwise} \end{array}$$

If $\text{detach}_{\mathfrak{u},\mathfrak{a}}(\mathcal{M}) = \mathcal{M}'$, then

$$\begin{array}{lcll} \mathcal{M}'(\mathfrak{a}) & = & u & \text{for some } \mathfrak{u}' \text{ where } \mathcal{M}^{-1}(\mathfrak{u}') = \emptyset \\ \mathcal{M}'(\mathfrak{b}) & = & m(\mathfrak{a}) & \text{if } \mathfrak{b} \neq \mathfrak{a} \end{array}$$

If $\text{move}_{\mathfrak{u},\mathfrak{u}',\mathfrak{a}}(\mathcal{M}) = \mathcal{M}'$, then

$$\begin{array}{lcll} \mathcal{M}'(\mathfrak{a}) & = & \mathfrak{u}' & \\ \mathcal{M}'(\mathfrak{b}) & = & \mathcal{M}(\mathfrak{b}) & \text{if } \mathfrak{b} \neq \mathfrak{a} \end{array}$$

These suggestions are evaluated using their *improvement* i.e. how much they make the cost lower.

The set of possible suggestions is then

$$\begin{array}{rcl} S & = & \{\text{combine}_{\mathfrak{u},\mathfrak{u}'} \mid \mathfrak{u}, \mathfrak{u}' \in \mathfrak{U}\} \cup \\ & & \{\text{detach}_{\mathfrak{u},\mathfrak{a}} \mid \mathfrak{u} \in \mathfrak{U} \wedge \mathfrak{a} \in \mathfrak{A}\} \cup \\ & & \{\text{move}_{\mathfrak{u},\mathfrak{u}',\mathfrak{a}} \mid \mathfrak{u}, \mathfrak{u}' \in \mathfrak{U} \wedge \mathfrak{a} \in \mathfrak{A}\} \end{array}$$

A modular structure $\mathcal{M}$ is *locally optimal* if there is no such suggestion $r \in S$ for which $\gamma(\mathcal{M}) > \gamma(r(\mathcal{M}))$.

For efficiency reasons, it is necessary to define when two *atoms are related*. The atoms $\mathfrak{a}$ and $\mathfrak{b}$ are related, if $(\overline{\mathfrak{a}}, \mathfrak{b}) > 0$ or $(\overline{\mathfrak{b}}, \mathfrak{a}) > 0$. *Modules* $\mathfrak{u}$ and $\mathfrak{u}'$ *are related* if there are such related atoms $\mathfrak{a}$ and $\mathfrak{a}'$ that $\mathfrak{a} \in \mathcal{M}^{-1}(\mathfrak{u})$ and $\mathfrak{a}' \in \mathcal{M}^{-1}(\mathfrak{u}')$. The best suggestions involving two modules are always between related modules.

Using the above concepts it is now possible to state how to find a locally optimal module structure. First let $\mathcal{M}_0$ be an initial module structure. One possible way is to find some suggestion $r \in S$ such that $\gamma(r(\mathcal{M}_0)) < \gamma(\mathcal{M}_0)$, and then repeat this procedure until there is no such suggestion. The suggestion r can for example be the first suggestion that was found or the best possible suggestion. A locally optimal module structure can be found faster, if several suggestions are performed at the same time. It is safe to perform suggestions for unrelated modules.

4.3.4 Significance based analysis

Assume there are n programmers working on a program that has modular structure $\mathcal{M}$. Each programmer performs m tasks. Then, the total system cost is $nm\gamma(\mathcal{M})$. If the cost for performing a change is x, the effort needed

for a programmer to accommodate the change is y, and the set of involved atoms is $\mathfrak{A}$, then the cost caused by performing the change is

$$x|\mathfrak{A}| + ny(1 - (1 - P(\mathfrak{A}))^m)|\mathfrak{A}|$$

The set $\mathfrak{A}$ can be calculated for refactoring r as

$$\mathfrak{A}(r, \mathcal{M}) = \{\mathfrak{a} \in \mathfrak{A} \mid r(\mathcal{M})(\mathfrak{a}) \neq \mathcal{M}(\mathfrak{a})\}$$

The total improvement for refactoring r is then

$$nm(\gamma(\mathcal{M}) - \gamma(r(\mathcal{M}))) - x|\mathfrak{A}(r, \mathcal{M})| - ny(1 - (1 - P(\mathfrak{A}(r, \mathcal{M})))^m)|\mathfrak{A}(r, \mathcal{M})|$$

This is quite easy to calculate, but now the system would need more parameters.

4.4 Quantitative analysis

The purpose of the following analysis is to determine what kind of features the cost function $\gamma(\mathcal{M})$ and the suggestion system have in practice. The evaluation proceeds in two phases. To understand better, how the proposed metric works, some more basic metrics are considered first. This helps in understanding what kind of software the package contains. Then the results produced by the cost function are analyzed, and we give a high-level analysis of how the suggestion system works. In Section 4.5 individual suggestions given by the system are analyzed.

4.4.1 Basic metrics

The tool was applied to a medium sized open source program, jEdit. The inspected package includes 6396 members. Of these, 715 are never used anywhere. One may assume that there are two factors that cause this phenomenon. First, plug-ins can be written in Java for jEdit, and second, the functionality of the editor can also be extended using a scripting language, which uses reflection to call members. Table 4.1 shows that most of the methods have less than four users, but some have very many users.

As discussed in Section 4.3, dynamic binding is taken into account so that each use has weight 1, which is divided between possible methods. Currently, only type information is used for resolving dynamic binding, but a points-to analysis might give more precise results. The distribution of weighted users can be seen in the middle of Table 4.1. The correlation between weighted and unweighted case is 0.66. This means that while the values are similar in general, there is a clear difference that is caused by weighting.

Next it was measured how many units use the members, see the right panel of Table 4.1. Occasions in which a unit uses its own members were

NCM	M
1	1824
2	1476
3	805
4	430
5	230
6	150
7	104
8	150
9	71
10	60
11-20	211
21-	143

WCM	M
0	699
1	1587
2	1390
3	744
4	399
5	216
6	136
7	96
8	71
9	61
10	44
11-20	145
21-	66

NCU	M
0	2765
1	1672
2	481
3	192
4	109
5	67
6	44
7	42
8	101
9	26
10	13
11-20	57
21-	85

Table 4.1: Left panel: histogram giving for each number of calling methods (NCM) the number of members (M). Middle panel: the same but the methods are weighted (WCM) based on dynamic binding. Right panel: unit users (NCU) are considered for members.

ignored. The results are collected into the right panel of Table 4.1. Members such as `toString` -methods are used by very many units. Most of the time, fields are not used by external units. This kind of metrics could be useful when determining in which order the IDE suggests completions for member names: members with most uses should be shown before rarely used members.

Table 4.2 shows an analysis of the number of members in the units. There are 275 compilation units. In the majority of these, there are more than 10 members.

Unit usage calculates, how many times the members of the units are used. This value basically represents the weight of the unit in our framework. More precisely, to calculate the weight of the unit, one has to take the weights of method calls into account. The unit usage and unit weight distributions are shown in the middle and right panels of Table 4.2.

4.4.2 Cost function and suggestions

The cost of modules was evaluated next. The coupling factor was set to $o = 30$. The total cost for all units was 1040491. Table 4.3 shows the largest costs for units. These units account for about half of the total cost. This indicates that the focus of our suggestion system should be in splitting these large classes into more manageable pieces.

NOM	Units
1	9
2	9
3	10
4	12
5	8
6	7
7	12
8	10
9	12
10	11
11-20	76
21	99

NCM	Units
0-	46
10-	26
20-	34
30-	20
40-	22
50-	12
60-	11
70-	14
80-	10
90-	9
100-	5
110-	39
210-	27

WCM	Units
0-	58
10-	40
20-	41
30-	25
40-	25
50-	16
60-	16
70-	8
80-	7
90-	4
100-	4
110-	17
210-	14

Table 4.2: Left panel: histogram giving for each number of members (NOM) the number of units. Middle panel: the number of members calling the units (NCM) is calculated. Right panel: the same, but the callers are now weighted (WCM).

The strategy described in Section 4.3 was applied to determine a locally optimal module structure. There are two alternatives for the initial module structure $\mathcal{M}_0$: One can start from the original module structure, or from singleton modules, where each atom initially has its own module.

Starting with original modules there were 4211 suggestions. When performing these, the total cost decreases from 1040491 to 485129. Starting from singleton modules, there are 5579 suggestions and the total cost decreases from 734532 to 458346. To get a high level idea, how the cost of the system changes, Figure 4.1 shows the measured system cost after performing each suggestion. After 38 suggestions, the improvement has been 100000 units. After 163 improvements, the cost has decreased 200000 units compared to the initial state, and after 388 suggestions, it has decreased another 100000 units. Even after 1000 suggestions, the remaining suggestions improve the system over 130000 units. This shows that in practice, it is impossible to approach a locally optimal module structure by applying individual suggestions. For best results, the system could be used from the beginning of the project. On the other hand, by performing only few suggestions, a large improvement can be achieved.

Figure 4.2 shows how much improvement the individual suggestions would make. After several steps have been made, the improvement of the suggestions lowers in general. However there are still suggestions that have

Unit	Cost	LOC
textarea.TextArea	173445	6059
jEdit	127822	4043
buffer.JEditBuffer	65570	2475
View	46765	1724
GUIUtilities	33047	1789
Buffer	30216	1998
browser.VFSBrowser	26658	1975
MiscUtilities	23760	1803

Table 4.3: Units with largest costs.

large improvements. It seems that large improvements are still possible after a local optimum has been found. Figure 4.3, illustrates that starting from singleton modules (each atom has its own module) makes some difference. The first bumps are caused by combining two atoms into one module, and the bumps at the end are caused by discrete values for improvements.

4.5 Qualitative analysis

When determining whether the proposed system is any good, one should keep in mind that in practice, the most important factor in software development is the informal knowledge of the developers. We call this the *subjective view*. Another factor is caused by the concrete software artifacts, the *objective view* to the software. Not all knowledge of the subjective view can be extracted from the objective view.

To get definite results on whether a suggestion system is good or not, we would need to have several similar projects; then a part of the projects would use the suggestion system, and another part of the projects would not use it. This setup seems to be impossible, since much resources would be needed for the projects, and the tool should also be mature enough.

Another alternative would be to experimentally measure how much time the programming tasks take. This would however not give definite results, since the total effect of the modular structure was not actually measured. A study that could for example show a strong correlation between the time for understanding a method and the time for modifying a method is also required. The setup would have to be constructed very carefully so that the result would not strongly depend on unknown factors. This kind of study is more related to program comprehension than the suggestion system itself. If this kind of study were performed, it could be used to improve our system by modifying the assumptions we made in Section 4.3.

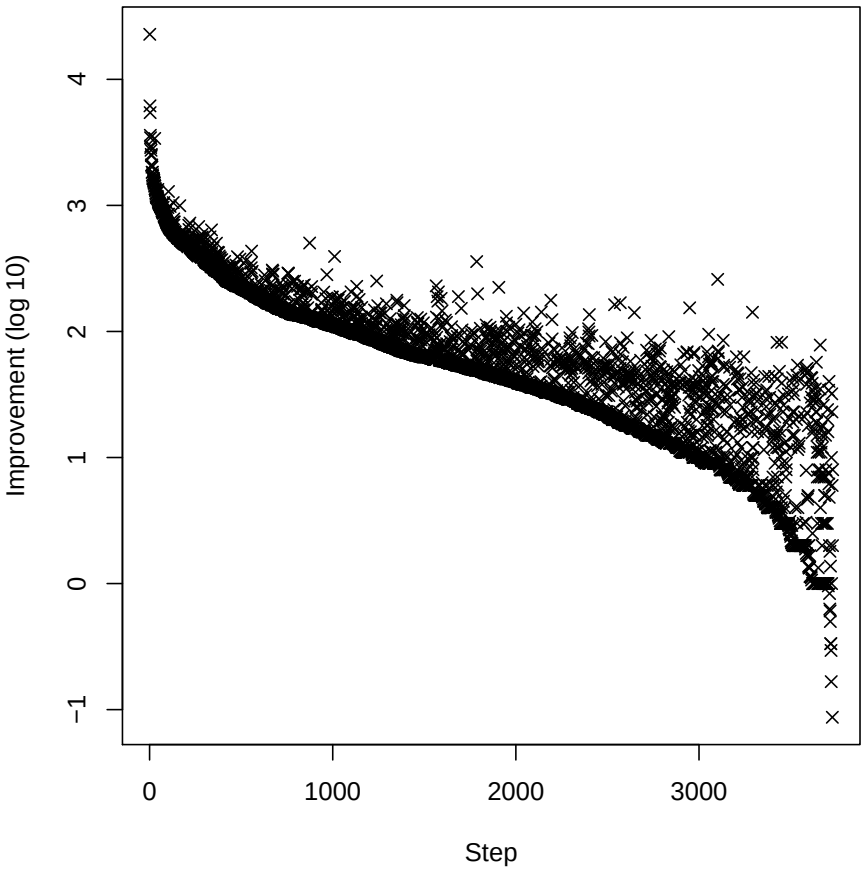

Figure 4.1: The change of the system cost when starting from the original system using $o = 30$.

Given that we have limited resources and the system is still quite immature, the following methodology is used to validate the proposed system:

- Identify the use cases for the system.
- For each use case, identify what kind of suggestions are desired.
- Analyze the suggestions given by the system against these criteria.

An issue is that the suggestions would now be analyzed against the design heuristics that can be shown to be included in the metric by construction. What makes the analysis interesting, is seeing how the system resolves conflicting heuristics in practice. Another point of interest is to determine, whether the subjective view is compatible with the results of our system. This issue is most evident when forming new modules.

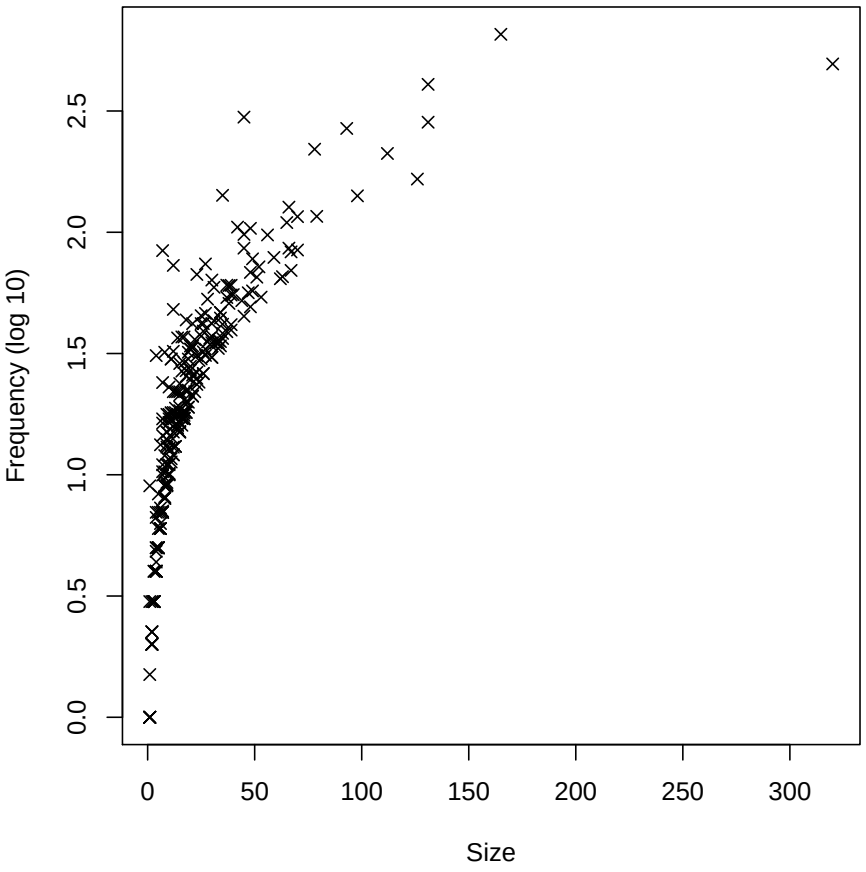

Figure 4.2: The amounts of improvements of steps, starting from the original system using $o = 100$.

The outcome of this study would be to identify practical problems in the system, and if the suggestions seem to have a good quality, a more matured tool should be implemented. The results of the study should give some ideas about how to further develop the system.

Quality of suggestions

There are several criteria that can be used for evaluating the quality of suggestions. One important requirement is that the suggestions should be easy to perform. Programming environments can perform several refactorings automatically. In Section 4.5.5 we show how to perform the refactorings suggested by our system.

A very important criterium is that the suggestions should have a large

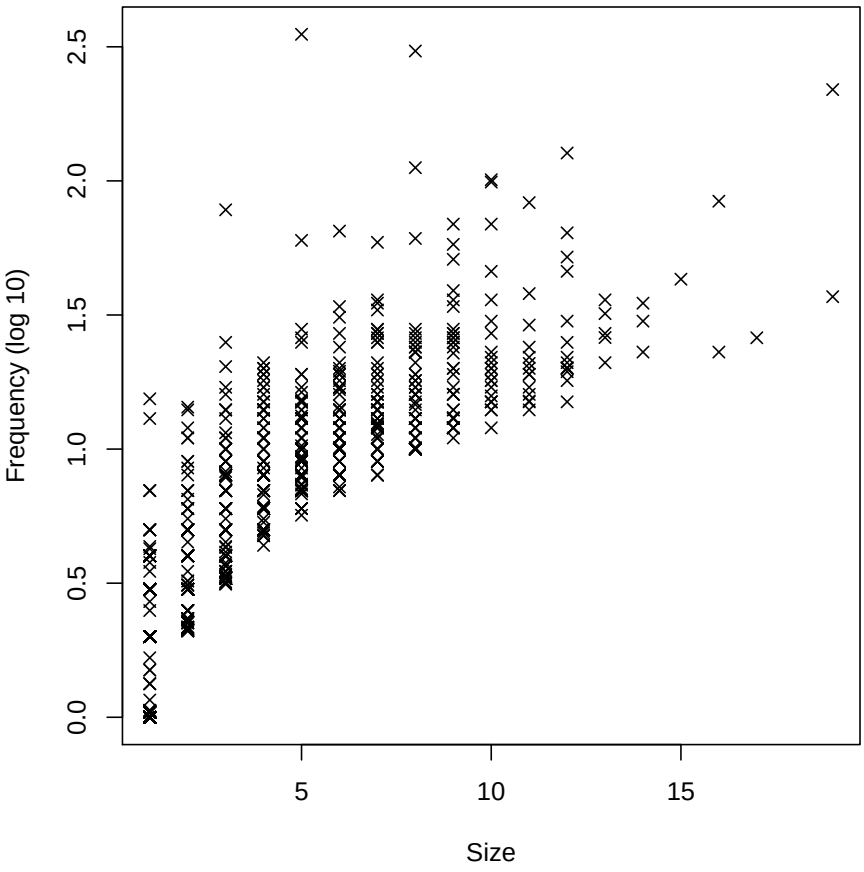

Figure 4.3: The amounts of improvements of steps, starting from singleton modules using $o = 10$.

effect. Current proposals for refactoring suggestion systems do not take this into account properly. If the threshold values are changed to be more strict, the system would find representative cases of design problems. However, the design problem might be in an insignificant part of the program. Even if the quality of the system is improved, the time taken by making the improvement might never be got back.

Another important criterion is that the reasoning behind the suggestions should be easy to understand. This is because eventually it is the programmer, who has to decide, whether the suggestion should be performed. To aid this task, visualization tools or other similar tools could be useful. It would also be helpful to have some kind of predefined categories for the reasons why the suggestions have been made. This kind of categories can hopefully be found by inspecting suggestions.

As another criterion, some kind of concept of reliability is needed for suggestions. For example, if an atom related to a suggestion has several suggestions with similar values, the reliability of the suggestion is not very good. Also, if a member is rarely used, the suggestions related to it might be unreliable, because the probabilities that are associated with it are not very reliable.

Use cases

On a basic level, the refactoring suggestion system has only one use case: Find design problems related to the modularity of the program. More specific use cases that we consider are:

- Finding the most problematic classes.
- Finding the best suggestions.
- Finding a locally optimal module structure.
- Breaking a large class into smaller pieces.
- Finding suggestions for moving members between modules.

Finding the most problematic classes was already considered in Section 4.4. Other cases are evaluated below.

4.5.1 Individual suggestions

The first 50 suggestions given by the system were investigated, and it was evaluated what kind of effects would performing them have.

Table 4.3 lists the classes with largest costs. As the first observation, most of the suggestions are related to these classes. Breaking these classes should be a priority in any suggestion system for modularity. Another observation about the importance of suggestions is that as can be expected, the first suggestions are about methods that are used very often. Also classes with lower cohesion get more important suggestions than more cohesive classes.

Most of the time, the members of the formed modules are closely related. The methods that are moved into the same module often have been grouped in the source code. Some suggestions move methods to related existing classes. These are effects that are caused by the notion of cohesion in our system. It is harder to see the effects of coupling in our system. In fact, the coupling of the system is slightly increased, because so many small modules are formed.

The first suggestions are related to the class `jEdit`. Three methods from this class, `getProperty`, `getBooleanProperty` and `getIntegerProperty`,

are moved to form a new module. These methods are used over 1000 times in the source code, and therefore they are essential for understanding and modifying the software. In fact the property-related methods are already grouped in the source code, but this grouping is done using the features of the jEdit-editor itself, and not using language features. This case is an example of an important property that the proposed system has: The module structure that is suggested by the system is similar to the programmer's intuition of the module structure of the program (although there are some clear differences). Another important property is that the proposed system rarely produces false positives, and the suggestions reflect real concerns in the module structure of the program.

One observation is that *call-chains* are detected. A call-chain means that an object is often called as $a.b().c()$. This should be replaced by $a.d()$, where d is a new method that calls b and c. The reason why call-chains are detected is that they create a close relationship between two methods in different classes. This should not be surprising, because already LCIC [60] detected call-chains.

An example of a detected call-chain is the method `getStatus` from class `View`. This method is often used in call-chains as the first called method. The suggestion is to move the method into class `StatusBar`, because this is the class that is returned by the method. The result would be that all status bar related methods would be in their own module. The problem with this is the fact that traditionally in OO design, the status bar is thought to be contained by the view, and therefore the relation is expressed as a field in the class that represents the view.

Another observation is that just as one can see from Figure 4.2, sometimes related members are moved in sequences. After the first related member has been moved, the next related members can have suggestions with very high values.

Some problems can also be observed. Large methods might cause unreliable relationships. Too small methods can cause similar problems. To make the effect of these methods smaller, lower probability could be given to the tasks associated with them.

Perhaps the worst problem with this use case is that often the intermediate moves are confusing. Several large classes have to be broken into smaller pieces, and the suggestions are interleaved, so it is hard to see what is happening. Also the formed modules can be modified later to form better modules. To solve this problem, the classes should be broken into pieces independently, or the locally optimal module structure should be used.

As a summary, the best suggestions are concentrated on important parts of the program, and they detect design problems, but they might be confusing to use.

4.5.2 Locally optimal module structure

One reason to inspect a locally optimal module structure is that as can be seen from Figure 4.1, it is impossible to get close to a locally optimal situation by performing individual suggestions. Also the suggestions might be confusing, because several classes are being broken at the same time.

Using the singleton modules, the refactorings were applied using the standard strategy until a locally optimal module structure was found. The generated modules usually fall into the following categories:

- Existing classes, or parts of them.
- Inner classes.
- Virtual methods with the same signature.
- Constructors and utility methods.

The most closely related members are the constructor for class `Log`, and fields `ERROR` and `DEBUG` from the same class. The second most related members are `endCompoundEdit` and `beginCompoundEdit` from class `JEditBuffer`. The relation between these two methods is so strong, that a programming error can be suspected if they are not used together.

The first closely related members that belong to different classes are `jEdit.getSettingsDirectory` and `MiscUtilities.constructPath`. This is an example of two related methods that have a different level of generality, since the method `constructPath` could be used by any program that uses the file system. The level of generality cannot be detected by the proposed refactoring suggestion system.

Often almost all members of the modules originate from one class. Additionally, there is often some related member from another class. The fact that the generated module structure has similarities with the original structure shows that the suggestion system captures something essential about the way programs are structured into modules. The explanation is that if a class is cohesive, the users of the data structured by the class want to use several fields of the data, thus generating relations between these fields. On the other hand, it is possible that some features of classes cause artificial relations between its members.

Virtual methods with the same signature are sometimes grouped into modules. This happens when the method signature is called many times, and there is not much relation with the other members in the class.

If a method is used rarely, it is often in the same module as its clients. This is not always the most logical solution, but the rarely used methods have also small weight in the system. Basically there is not enough evidence to determine where they belong.

Strongly coupled classes are usually broken into modules, which have members from the coupled classes. This kind of modules might implement one functionality. This means that the involved classes have stronger coupling than cohesion.

As a summary, the most important observation is that the classes formed by the suggestion system follow the original structure of the program. This shows that the forming of modules is not purely based on subjective or external factors. Locally optimal modules are probably not too useful otherwise. Instead, the classes can be broken separately.

4.5.3 Breaking a class

To evaluate how breaking individual classes works, the classes `TextArea`, `jEdit` and `View` were inspected carefully. Again, singleton methods were used as a starting point, and a locally optimal module structure was found using the standard strategy.

The core functionality of the classes (if exists) forms the *core module* of the original class. The core module is also the largest module. This is because in general, the most important members are in larger modules, and less important members are in smaller modules. For example the class `TextArea` has a core module with 18 members containing fields `caret`, `caretLine` and `buffer`.

Members that have no users are often in singleton modules. The idea is that these members are then hidden when they are not used.

If the coupling factor is large, the related functionality is not always combined. The suggested modules are smaller than what would seem reasonable from the subjective viewpoint. The first possibility is that the subjective viewpoint is wrong and the modules should be very small. Another possibility is that some of the assumptions can be modified so that larger modules would be generated. For example, we could add vector components for each original module. The last possibility is that purely subjective factors have so large impact that the system can never be improved significantly. At least the found modules can be used as a starting point for breaking the class.

One issue that weakens the reliability of the suggestions is that there are accessor methods for the fields. This is because the accessor methods and the fields can be used interchangeably. The accessor methods can be thought to be too small methods for the reliability of the suggestion system. As an additional observation, these accessor methods can form their own modules. This is an example of layers in the module structure. In general, the suggestion system does not take layers into account.

Almost always the formed modules contain members with related functionality. A larger piece of functionality can be divided into several modules. Sometimes there are several methods that implement similar functionality.

These methods might not be in the same module because another one is redundant.

If there are some kind of usage patterns for members of the class, they generate strong relationships between the members in the patterns.

A problem with breaking classes is that it is not clear what the most important suggestions are. For example, there might be an obvious way to break an average sized class, but this might not be found first. Instead, the best suggestion would probably be for the largest classes.

Because of the complexity of the relationships, visualization could help in breaking a class at least in the following ways

- See how close the methods are to other methods.
- See how close the modules are to methods.
- See how close modules are to each other.

As a summary, the system can be used to find cohesive parts in the classes. These parts are in line with the intuition of the programmer. The found parts can be used to divide the class into smaller modules to make the software significantly easier to understand and learn.

4.5.4 Intermodule suggestions

The remaining use case is finding members that should be moved into other modules, that is, finding interclass modifications. A difficulty in this lies in separating the interclass modifications from intraclass ones. We first broke all classes using the method described in the previous use case. Then we checked for the suggestions for the whole system. It was observed that this gave a better local optimum than starting from singleton modules, or from original modules directly.

One common reason for intermodule suggestions is the existence of call chains. The call chains are now detected more clearly than when considering the best suggestions overall, because the size of the modules is not an issue anymore. Another similar possibility is that a method a is always called with $a(b(x))$. In this case, the call to b should be moved into a. These two examples show that there are other refactorings than Move member that should be considered for improving the module structure.

Closely related members are combined into one module. This kind of modules are related to some functionality. Because the classes were first broken into small pieces of functionality, there might be some kind of vertical properties, where some functionality, whose implementation was divided into several classes is now moved into one module.

A member can be moved to a closely related module. This happens when the functionality of the member is more closely related to the functionality

of the new module than to the functionality of the old one. For example the Feature Envy code smell can cause this kind of situation.

A problem is that sometimes a member is more general than the program, i.e. it could be used in some other program, too. The probability of this kind of members cannot be estimated correctly by the system. Because of this, a member that is general, can be moved into a module that should not have so general members.

The proposed system gives very good suggestions for interclass refactorings, but there is one problem. The improvement of the cost function is small compared to the improvement that is gained by breaking the classes.

4.5.5 Implementing refactorings

Sometimes it might be hard to immediately see how a member can be moved into another module. One of the reasons is that in programming language design, little or no attention is given for making programs easier to modify or refactor. Perhaps the opposite is true: The language is supposed to enforce a good, object-oriented style. In practice this means that a large number of classes are either (i) tiny helper classes that are needed to overcome the limitations of the OO paradigm, or (ii) huge classes with tons of methods and fields.

A generic way to move fields is simple enough: Just make a static variable that is a mapping from objects to the type of field. Moving methods is more difficult because of dynamic binding. A new movable concept is needed that represents the method signature. This kind of implementation for the proposed refactorings might sometimes be cumbersome and lead to ugly looking code.

As an alternative for moving a method into another module, perhaps the method should be broken into two pieces, and only the other part of the method should be moved, then.

A part of the class can be quite easily extracted into a new class. Then a reference can be added into the original one. The problem is that if the refactoring is performed many times, the original class might become a collection of fields with low cohesion.

Eliminating call chains should be used in applicable situations. In the case where a method is only used in a call chain, the chained call can be moved into this method.

Sometimes the system suggests creation of a module, where there are several methods with same signature. If the module seems reasonable, the visitor pattern can be used to implement it.

As a summary, it is possible to implement the suggested refactorings. Sometimes implementing them is cumbersome, and better alternatives should

be used. A more fine-grained suggestion system should help with these issues.

4.6 Conclusions

The concept of refactoring metrics bridges the gap between metrics and refactorings. Our results indicate that refactoring metrics can be used as a basis for a refactoring suggestion system.

One important property of the proposed metric is that the locally optimal module structures found using the metric seemed to be quite reasonable, and often followed the original structure. This was achieved because the metric actually attempts to approximate the role of modules in the software process. It is unlikely that a metric that approximates a single design heuristic would have the same property.

In general, the quality of examined suggestions seems to be good. More investigation is needed to establish this. For example, one could measure the number of generated modules with different sizes, and perform a more detailed analysis of how similar the generated module structure is to the original structure developed by programmers. Also automatic categorization of the suggestions would be useful.

More comparisons with other approaches should be made. Compared to the threshold based approach, the advantage is that our system can distinguish which suggestions are the most important ones. The disadvantage might be that it is harder to accommodate new refactorings in our system. The research for refactoring suggestion systems is still new, and there are some obvious approaches that have not yet been tried. For example, linear regression has been used to approximate some empirical value-based on metrics [13]. The result from linear regression could be used as the cost function for a refactoring suggestion system.

A way to extend the scope of the refactoring suggestion system would be studying different or more complex notions of tasks. Focusing on programming operations means that also the programming environment (IDE) should be taken into account when defining metrics. The reason for this is that the programming environment includes tools for refactoring and program comprehension. These tools are related to the cost of refactoring and programming tasks. Monitoring programmer behavior can be used to gather empirical information about these tasks [82].

One of the most likely directions for better tool support in software development is model driven development. We envision that software models will have a larger role in the future of software metrics. For example, it needs to be evaluated, whether traditional design heuristics are any good when these models are used. It is also possible that different models and

tasks are relevant on different phases of a software process, for example, different kinds of models should be used in design and maintenance phases.

The module structure considered in our system should be regarded as some kind of software model. This model represents a very raw form of modularity. The system can be extended by considering more advanced concepts of modularity, such as layers, interfaces and information hiding. Refactorings should be defined to reveal the software engineering properties of these software artifacts.

The most pressing issue that needs to be addressed in further research is accommodating refactorings such as Remove Call Chain. These refactorings are clearly related to the modular structure of the programs, even though they do not directly modify it. One possible approach for this issue is as follows: Similarly as we used Move Member -refactorings to define the modular structure, we now use other refactorings to define other models. Then cost functions can be defined for these models. Once these models have been defined, they can also be used for refactoring of programs. Techniques from model driven development can be used to synchronize models and the source code.

Chapter 5

Conclusions

This dissertation had three different parts dealing with internal cohesion, external cohesion, and a refactoring metric for modularity.

It turned out that internal cohesion can not be reliably used as a reliable design heuristic; we found a number of programming patterns that were responsible for bad internal cohesion. Then we turned to external cohesion, which was shown to be a better indicator of program quality.

A problem with external cohesion metrics was that further analysis is needed to identify the actual design problems. This problem was solved by introducing refactoring metrics, which produce concrete suggestions for the programmer. Our analysis shows that the proposed refactoring metric produces reliable suggestions. Practical tests with it indicated that there still remain some possibilities to further development of the metric so that it matches even better with programmer intuition.

When developing refactoring metrics, two topics for further studies were recognized:

- Developing new technologies that transform programs from being collections of text files to more structured pieces of data. This is related to model driven engineering.
- Finding out what kind of changes are made for programs. This would help us to find out what kind of operations the development environment should support.

The contributions of the present study can be divided into two categories: practical and scientific. On the practical front, the following issues are relevant:

- Software analysis tools. A set of tools were developed, but the most important was the use of off-the-shelf tools like XQuery. To make program analysis mainstream, easy to use file or database formats are necessary.

- Mathematical notations. Compact mathematical notations have to be developed, so perhaps one day software engineering will live up to its name.

- Software product metrics. The metrics developed here should be useful in monitoring and improving the modularity of software.

Let us sketch two opposite approaches for producing knowledge about software engineering. The first is to consider the process of writing software with an experimental approach. For example it could be monitored how students make software in a controlled environment. The second is to consider the software as a historical artifact. In this approach for example the Linux source code could be important, and the software made by students has zero significance. The first approach is probably useful for research about software process or teamwork, and the second is more relevant to software product metrics.

Concerning internal cohesion of programs, the following finding was most important: Given the number of fields and methods, and number of connections between them, each cohesion graph has equal probability. (Recall that cohesion graph contains the relationships between fields and methods in a class.) To the extent this statement is true, it is impossible for internal cohesion metrics to tell anything useful about software quality. Other important findings related to internal cohesion were related to the concept of design anomaly and discovering design rules.

Concerning external cohesion, it was found out that the module structure of programs can be derived from the internal relations in the program and the properties of the development environment. This finding is important, because otherwise it could have been argued that the module structure depends on the domain model, and cannot be evaluated using software product metrics.

To sum up, we found principles (size, cohesion, coupling and significance) that govern the modularity and our theory explains the reason these principles are valid. This may eventually lead to a new methodology to create programs that are guaranteed to have good quality.

Chapter 6

Recent work on software cohesion metrics

In this chapter we give an overview of the new work that has been published since finishing our research in Chapters 2, 3 and 4.

Several studies have focused on finding new ways to empirically evaluate software metrics and to show their relation to software quality. Among others Arora [7] et al. study the quality of Java library classes. The metrics that they consider are CK and MOOD metrics sets. In the study of Jassim and Altaani [51] the statistical properties relating to metrics are considered using linear regression.

Radjenović et al. [80] have made an extensive review of attempts to relate software metrics to software faults. In this study, cohesion metrics were not found to be particularly effective in predicting faults. Another similar study was made by Jabangwe et al. [50]. They considered other quality attributes in addition to fault prediction. Cohesion metrics were found to be quite effective in this work. Also in the study of Goyal, Sandra and Singh [45], where interaction between metrics was considered, it was found that LCOM was influential in interaction with other variables. By interaction it is meant that if there are two variables x and y, then the model used for linear regression would be $\alpha + \beta x + \gamma y + \delta xy$. The term xy is the interaction term. He et al. [48] attempted to find a minimal set of metrics for fault prediction. The set they ended up with was LOC, CBO and LCOM.

Machine learning techniques can be used to improve metrics based fault prediction. For example Rodrigues et al. [83] used subgroup discovery to generate rules for fault predictions. Basically this technique can be used to find rules that are similar to the kind proposed by Lanza [56]. The metrics suite used by them was CK.

The fact that several studies mostly consider metrics that have been

defined before year 2000 shows that it is hard for new metrics to gain popularity despite the efforts to develop better metrics and to show that they have better properties than the previous ones. From this it would appear that not much progress has been made on the field of software metrics in past two decades.

Many of the current studies follow the same ideas as before. For example new formulae are proposed for calculating the cohesion graph, or a more procedural cohesion based on concept of slices are proposed. A slice of a procedure is a part that is related to a certain variable. For example a forward-slice is the part of a procedure that depends on an argument, and a backwards-slice is the part that is related to a return value. Based on these slices, one can define a more fine-grained cohesion measure between variables and methods. For example Yang et al. [92] have studied this kind of metrics. Their results show that slice based metrics are statistically independent from other cohesion metrics. On the other hand, metrics that are defined by just taking the square-root of an old metric would be statistically independent, so it remains an open question what is the additional information gained by these types of metrics.

Some work has been made to use metrics for software quality improvement. One possibility is applying the metrics at package level. For example Abdeen et al. [1] proposed new cohesion and coupling metrics for packages. These metrics are quite similar to metrics that have been proposed for classes. In a follow up [2], these metrics were applied to improve the quality of modularity in the program. A search for better modular structure was made based on a combination of existing metrics. A more elegant approach would have perhaps been to define a metric that evaluates the quality of modularity.

Another effort that considers refactorings was by O'Cinneide et al. [27]. Effects of refactorings into cohesion metrics were considered in that work. It was observed that when performing refactorings related to modularity, values of different cohesion metrics change to different directions. Among the cohesion metrics they considered were LSCC, LCOM and TCC.

Al Dallal [30] has made research on the impact of inheritance on cohesion metrics. He has also considered the impact of inheritance of variables and methods separately. Another topic considered by Al Dallal was to improve the applicativity of cohesion metrics, that is, to change the definitions of metrics so that more classes get metrics values.

Al Dallal and Briand [3] defined a metric LSCC based on similarity of methods. The notion of similarity they considered was how many accessed variables they share. The authors also considered the relation of refactorings to their metric. The definition of LSCC is as follows. Assume the class has a set of methods $\mathfrak{M}$ and set of fields $\mathfrak{F}$, and a relation $R(\mathfrak{m}, \mathfrak{f})$ holds when $\mathfrak{m}$

uses field $\mathfrak{v}$. Then the similarity of methods $\mathfrak{m}_1$ and $\mathfrak{m}_2$ is

$$\mathrm{sim}(\mathfrak{m}_1, \mathfrak{m}_2) = \frac{|R(\mathfrak{m}_1, \mathfrak{F}) \cap R(\mathfrak{m}_2, \mathfrak{F})|}{|\mathfrak{F}|}$$

LSCC for the class is the average of similarity for all pairs $\{\mathfrak{m}_1, \mathfrak{m}_2\} \subseteq \mathfrak{M}$:

$$LSCC = \sum_{\{\mathfrak{m}_1, \mathfrak{m}_2\} \in \mathfrak{M}} \frac{\mathrm{sim}(\mathfrak{m}_1, \mathfrak{m}_2)}{|\{S \subseteq \mathfrak{M} \mid |S| = 2\}|}$$

Al Dallal and Briand also defined a similar cohesion metric called HSCC. This metric applies to UML diagrams instead of source code.

An interesting new metric has been proposed by Drouin, Badri, and Toure [31]. Their metric Quality Indicator (Qi) attempts to give a value that approximates the overall quality of a class. Let $P(\mathfrak{m}_1 \ldots \mathfrak{m}_n \mid \mathfrak{m})$ denote the probability that when method $\mathfrak{m}$ is called, the method $\mathfrak{m}$ directly calls methods $\mathfrak{m}_1 \ldots \mathfrak{m}_n$ in the given order. The probabilities are assigned in such a way that in an if-statement each branch is considered to have equal probability, and in loops, the probability of exiting the loop is $1/4$. Then Qi is defined by a system of equations, where for each method there is an equation

$$Qi(\mathfrak{m}) = Qi^*(\mathfrak{m}) \sum P(\mathfrak{m}_1 \ldots \mathfrak{m}_n \mid \mathfrak{m}) Qi(\mathfrak{m}_1) \cdots Qi(\mathfrak{m}_n)$$

The system of equations is solved iteratively. $Qi^*(\mathfrak{m})$ is the intrinsic quality assurance indicator, and it is calculated as

$$Qi^*(\mathfrak{m}) = 1 - \frac{1 - \mathrm{tc}(\mathfrak{m})\mathrm{cc}(\mathfrak{m})}{\max \mathrm{cc}(\mathfrak{M})}$$

Here $\mathrm{cc}(\mathfrak{m})$ is cyclomatic complexity of method $\mathfrak{m}$ and $\mathrm{tc}(\mathfrak{m})$ is *test coverage percentage*. Test coverage percentage is the percentage of paths that are covered by the automated tests related to the method. The Qi value of a class is the product of the Qi values of its methods. The authors found out that the value of the metric was quite stable when the software evolved, compared to other metrics that varied a lot more.

One trend is adding annotations to code to mark which parts of the code implement which feature of a program. Apel and Beyer [6] show how cohesion metrics can be defined for these features. Their metrics use distance based clustering of features. Olszak and Jørgensen [73] consider remodularization of programs based on features.

Qu et al. [79] used the concept of modularity from networking theory, and applied it to call graphs. The program was grouped into modules, and then a cohesion metric for classes was defined based in these modules.

A problem with this approach is that it is not immediately clear how the concept of modularity in networking theory is related to quality of programs.

Fokaevs et al. [41] considered Extract Class refactorings. The approach they used is distance based clustering.

Appendix A

Mathematical symbols

Symbol	Pronunciation	Meaning
$\mathfrak{a}, \mathfrak{b} \in \mathfrak{A}$	A	Atom in Chapter 4, otherwise any set.
$\mathfrak{c}, \mathfrak{d} \in \mathfrak{C}$	C	Classes.
$\mathfrak{e}$	E	Expression.
$\mathfrak{f} \in \mathfrak{F}$	F	Fields.
$\mathfrak{l}$	L	Location.
$\mathfrak{m} \in \mathfrak{M}$	M	Methods, in Chapter 3 includes fields.
$\mathcal{M}$	M	Modules.
μ	mu	Metric.
$\mathfrak{p} \in \mathfrak{P}$	P	
$\mathfrak{s} \in \mathfrak{S}$	S	Method signatures.
$\mathfrak{t} \in \mathfrak{T}$	T	Types.
$\mathfrak{u} \in \mathfrak{U}$	U	Compilation units.
$\mathfrak{v} \in \mathfrak{V}$	V	Views.

Bibliography

[1] Hani Abdeen, Stéphane Ducasse, and Houari Sahraoui. Modularization metrics: Assessing package organization in legacy large object-oriented software. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 394–398. IEEE, 2011.

[2] Hani Abdeen, Houari Sahraoui, Osama Shata, Nicolas Anquetil, and Stéphane Ducasse. Towards automatically improving package structure while respecting original design decisions. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 212–221. IEEE, 2013.

[3] Jehad Al Dallal and Lionel C Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(2):8, 2012.

[4] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[5] H. Aman, K. Yamasaki, H. Yamada, and M.-T. Noda. A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts. In T.Welzer et al., editor, *Proc. of Fifth Joint Conference on Knowledge-based Software Engineering*, pages 102–107. IOS Press, 2002.

[6] Sven Apel and Dirk Beyer. Feature cohesion in software product lines: an exploratory study. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 421–430. IEEE, 2011.

[7] Deepak Arora, Pooja Khanna, Alpika Tripathi, Shipra Sharma, and Sanchika Shukla. Software quality estimation through object oriented design metrics. *IJCSNS International Journal of Computer Science and Network Security*, 11(4), 2011.

[8] L. Badri and M. Badri. A Proposal of a New Class Cohesion Criterion: An Empirical Study. *Journal of Object Technology, Special issue: TOOLS USA 2003*, 3(4):145–159, April 2004.

[9] Dirk Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006, Shanghai, May 20-28)*, pages 807–810. ACM Press, New York (NY), 2006.

[10] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems. In *IWSM '00: Proceedings of the 10th International Workshop on New Approaches in Software Measurement, LNCS 2006*, pages 1–17, London, UK, 2000. Springer-Verlag.

[11] James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. In *ACM SIGSOFT Symposium on Software Reusability*, pages 259–262, 1995.

[12] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring - improving coupling and cohesion of existing code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 144–151, Washington, DC, USA, 2004. IEEE Computer Society.

[13] L.C. Briand, J. Wüst, and H. Lounis. Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. *Empirical Software Engineering*, 6(1):11–58, 2001.

[14] Lionel Briand, Khaled El Emam, Sandro Morasca, Khaled El, Emam Sandro Morasca, Centre De Recherche Informatique De, Centre De Recherche Informatique De, and Piazza L. Da Vinci. Theoretical and empirical validation of software product measures. Technical report, ISERN-95-03, International Software Engineering Research Network, 1995.

[15] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[16] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.

[17] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Measuring and Assessing Maintainability at the End of High Level Design. In *Proceedings of Internaltional Conference on Software Maintenance*, pages 88–97, 1993.

[18] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and validating high-level design metrics. Technical Report CS-TR-3301, Department of Computer Science, University of Maryland, College Park, MD, 20742, June 1994.

[19] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, 1998.

[20] Sio Bryton and F. Brito e Abreu. Modularity-oriented refactoring. In *Proceedings of the 12th European Conference on Software Engineering and Reengineering*, pages 294–297. IEEE Computer Society, 04 2008.

[21] H.S. Chae and Y.R. Kwon. A Cohesion Measure for Classes in Object-Oriented Systems. In *Proc. of the Fifth International Software Metrics Symposium*, pages 58–166, 1998.

[22] H.S. Chae, Y.R. Kwon, and D.H. Bae. A Cohesion Measure for Object-Oriented Classes. *Software - Practice & Experience*, 30(12):1405–1431, 2000.

[23] H.S. Chae, Y.R. Kwon, and D.H. Bae. Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables. *IEEE Transactions on Software Engineering*, 30(11):826–832, November 2004.

[24] Z.-Q. Chen, B.-W. Xu, and Y.-M. Zhou. Measuring class cohesion based on dependence analysis. *Journal of Computer Science and Technology*, 19(6):859–866, 2004.

[25] S.R. Chidamber and C.K. Kemerer. Towards a Metric Suite for Object Oriented Design. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91), Sigplan Notices 26(11)*, pages 197 – 211, 1991.

[26] S.R. Chidamber and C.K. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[27] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *ESEM*, pages 49–58, 2012.

[28] L.L Constantine and E. Yourdon. *Structured design*. Prentice Hall, 1979.

[29] G.W. Cox, L.H. Etzkorn, and W.E. Hughes. Cohesion Metric for Object-Oriented Systems Based on Semantic Closeness from Disambiguity. *Applied Artificial Intelligence*, 20(5):419–436, 2006.

[30] Jehad Al Dallal. The impact of inheritance on the internal quality attributes of java classes. *Kuwait journal of science & engineering*, 39, 2012.

[31] Nicholas Drouin, Mourad Badri, and Fadel Touré. Analyzing software quality evolution using metrics: An empirical study on open source software. *Journal of Software*, 8(10), 2013.

[32] `http://www.eclipse.org/`.

[33] Michael Eichberg, Daniel Germanus, Mira Mezini, Lukas Mrokon, and Thorsten Schr. QScope: an Open, Extensible Framework for Measuring Software Projects. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 113 – 122. IEEE Computer Society, 2006.

[34] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[35] Michael English, Jim Buckley, and Tony Cahill. Fine-grained software metrics in practice. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 295–304, Washington, DC, USA, 2007. IEEE Computer Society.

[36] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Foundations of Software Science and Computation Structures*, pages 14–30. Springer, 1999.

[37] L. Etzkorn, C. Davis, and W. Li. A Practical Look at the Lack of Cohesion in Methods Metric. *Journal of Object-Oriented Programming*, pages 27–34, September 1998.

[38] Willliam Feller. *An introduction to probability theory and its applications*, volume 2. John Wiley & Sons, 2008.

[39] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[40] L. Fernández and R. Pe na. A Sensitive Metric of Class Cohesion. *Information Theories and Applications*, 13(1):82–91, 2006.

[41] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[42] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.

[43] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[45] Rinkaj Goyal, Pravin Chandra, and Yogesh Singh. Identifying influential metrics in the combined metrics approach of fault prediction. *SpringerPlus*, 2(1):627, 2013.

[46] Mark Grand. *Patterns in Java, volume 1: a catalog of reusable design patterns illustrated with UML.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[47] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[48] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.

[49] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1996.

[50] Ronald Jabangwe, Jürgen Börstler, Darja Šmite, and Claes Wohlin. Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review. *Empirical Software Engineering*, 20(3):640–693, 2013.

[51] Firas Jassim and Fawzi Altaani. Statistical approach for predicting factors of mood method for object oriented. *CoRR*, abs/1302.5454, 2013.

[52] Padmaja Joshi and Rushikesh K. Joshi. Microscopic coupling metrics for refactoring. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 145–152, Washington, DC, USA, 2006. IEEE Computer Society.

[53] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 20, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[55] Ralf Lämmel. Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 15–28. ACM, 2002.

[56] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[57] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 2008.

[58] Sami Mäkelä and Ville Leppänen. Taking Purpose of Class into Consideration in Cohesion Metrics. In Varmo Vene Merik Meriste, editor, *Proceedings of the Ninth Symposium on Programming Languages and Software Tools*, pages 112 – 125, 2005.

[59] Sami Mäkelä and Ville Leppänen. Observation on Lack of Cohesion Metrics. In *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech'06)*, 2006.

[60] Sami Mäkelä and Ville Leppänen. A Software Metric for Coherence of Class Roles in Java Programs. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007*, pages 51–60. ACM, 2007.

[61] Sami Mäkelä and Ville Leppänen. Client based Object-Oriented Cohesion Metrics. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 743–748, Washington, DC, USA, 2007. IEEE Computer Society.

[62] Sami Mäkelä and Ville Leppänen. Client-Based Cohesion Metrics for Java Programs. *Science of Computer Programming*, 72(5-6):355–378, 2009.

[63] Sami Mäkelä and Ville Leppänen. Experimental Evaluation of Interpretations for Local Cohesion Metrics. In *Proceedings of the 2009 International Conference on Software Engineering Research and Practice(SERP'09)*, pages 389–395. CSREA Press, 2009.

[64] A. Marcus and D. Poshyvanyk. The Conceptual Cohesion of Classes. In *Proceedings, 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142, 2005.

[65] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1):29–98, 2000.

[66] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[67] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, second edition, 1997.

[68] Bertrand Meyer. Object oriented software construction. 1988.

[69] Vojislav B. Misic. Cohesion is structural, coherence is functional: Different views, different measures. In *IEEE METRICS*, pages 135–, 2001.

[70] Vojislav B. Misic. Measuring the coherence of software product line architectures. In *Software Engineering Research and Practice*, pages 364–372, 2003.

[71] Helmut Neukirchen, Benjamin Zeiß, Jens Grabowski, Paul Baker, and Dominic Evans. Quality assurance for TTCN-3 test specifications. *Software Testing, Verification and Reliability (STVR)*, 18:71–97, June 2008.

[72] Alexis O'Connor, Macneil Shonle, and William Griswold. Star diagram with automated refactorings for Eclipse. In *Eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2005. ACM Press.

[73] Andrzej Olszak and Bo Nørregaard Jørgensen. Remodularizing java programs for improved locality of feature implementations in source code. *Science of Computer Programming*, 77(3):131–151, 2012.

[74] L. Ott and J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the First IEEE-CS International Software Metrics Symposium*, pages 78–81, 1993.

[75] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[76] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[77] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[78] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003.

[79] Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Lidan Wang, Yuqiao Hou, and Zijiang Yang. Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software*, 108:193–210, 2015.

[80] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[81] Jan Rittinger, Jens Teubner, and Torsten Grust. Pathfinder: A Relational Query Optimizer Explores XQuery Terrain. In *Proceedings of BTW Conference (Datenbanksysteme fr Business, Technologie und Web)*, pages 617–620, 2007.

[82] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electron. Notes Theor. Comput. Sci.*, 166:93–109, 2007.

[83] Daniel Rodriguez, Roberto Ruiz, Jose C Riquelme, and Rachel Harrison. A study of subgroup discovery approaches for defect prediction. *Information and Software Technology*, 55(10):1810–1822, 2013.

[84] Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 37, Washington, DC, USA, 2002. IEEE Computer Society.

[85] Arto Salomaa. *Formal languages*. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[86] Santonu Sarkar, Avinash C. Kak, and N S. Nagaraja. Metrics for analyzing module interactions in large software systems. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 264–271, Washington, DC, USA, 2005. IEEE Computer Society.

[87] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[88] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics Based Refactoring. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38, 2001.

[89] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 39, pages 315–330, New York, NY, USA, October 2004. ACM Press.

[90] `http://www.uml.org/`.

[91] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and Binary Decision Diagrams for Program Analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 97–118. Springer-Verlag, November 2005.

[92] Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, Baowen Xu, Hareton Leung, and Zhenyu Zhang. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study. *Software Engineering, IEEE Transactions on*, 41(4):331–357, 2015.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

ISBN 978-952-12-3388-3
ISSN 1239-1883

Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences