University of Oslo
Department of Informatics

Dynamic Coupling Measurement for Object-Oriented Software

Audun Føyen

Cand. Scient. Thesis

29th January 2004

Abstract

A major goal of software engineering research is to develop techniques, methods and tools that may improve software quality. This thesis contributes to that goal.

It is possible to assume two different views on quality as it relates to software products. In the external view, quality is determined based on how well a product performs in practise, i.e., maintainability and usability. In the internal view, quality is derived from attributes inherent in the software product, e.g., structural properties such as coupling, cohesion and size.

Much research related to software quality models has focused on establishing relationships between structural properties and external quality attributes. The ultimate goal of this research is to develop quality prediction models, which may aid in making informed decisions concerning, for example, refactoring or program design.

Regardless of the structural properties considered, most quality prediction models have so far been based on static analysis of source code or designs. Such models have proven to be fairly accurate on some occasions. However, in the context of object-oriented systems, static coupling measures may not always be accurate, thus resulting in unreliable prediction models. Due to polymorphism and dynamic binding, static coupling measures do not always reflect the actual coupling taking place between classes, as this can only be determined at run-time. In addition, static measurements of coupling may be inaccurate when obtained from systems containing "dead" code.

In an attempt to overcome these problems, twelve dynamic coupling measures have been proposed. They differ from static coupling measures in that they are based on analysis of the actual messages exchanged between objects at run-time. The twelve measures are therefore referred to as "dynamic coupling measures". To collect the dynamic coupling measures, a tool called Jdissect was developed. Jdissect collects data from running Java programs to calculate dynamic coupling.

There are three objectives for the investigation of the proposed coupling measures. The measures need to be theoretically validated, that is, one needs to assess their theoretical properties and validity as coupling measures. Furthermore, it is important to determine whether they provide data over and above what can be collected through static measures such as size and static coupling. Finally, to demonstrate practical usefulness of the dynamic coupling measures, they must be evaluated as predictors of external quality. In the case study presented in this thesis, the external quality attribute considered for the evaluation is change proneness, which is an indirect measure of software maintainability.

The results indicate that some of the dynamic coupling measures are strong indicators of change proneness and that they complement existing static measures. The resulting prediction models may, for example, be useful to focus restructuring efforts on those parts of the software that are predicted to be the most likely to undergo future changes.

Acknowledgements

First of all, I am grateful to my supervisor Erik Arisholm for his guidance and enthusiasm. Our many long meetings and brainstorming sessions have taught me a lot. Furthermore, the opportunity to take part and assist in his research inspired me to no end.

I also wish to thank Lionel Briand for his help and for some very interesting discussions during the course of this project.

Additionally, there are a number of people who have provided assistance by taking the time to read and debate my thesis. In no particular order, I wish to thank Hilde Skjevling, Jørn Grotnes, Ragnar Nicolaysen, Christian Brinch and Christian Herzog.

Oslo, January 2004
Audun Føyen

Contents

Chapter 1

Introduction

Just four decades ago the concept of large scale software development projects was practically unheard of. Reports on the complexity of such projects started appearing in the '70s with works such as Brooks' "The Mythical Man Month" [BJ95], receiving widespread readership and distribution. The problems reported by Brooks' are still relevant, as seen in the Chaos Report, published in 1994 [SG94]. Software projects still tend to take more time and cost more money than estimated. That is, if they ever finish at all. The economic ramifications for both companies and organisations are potentially severe.

Consequently, a major goal of software engineering research is to develop and evaluate methods, tools and techniques that will improve the quality of software while reducing the cost commonly associated with it. This chapter describes the context, motivation and the main contributions of this thesis, thus explaining how we contribute to that goal.

Overview

The first part of this chapter is organised as follows: Section 1.1 provides some motivation for empirical research in software engineering. Common approaches to research in this field are presented in Section 1.2, followed by a summary of frequently seen forms of criticism against them. Together, the discussion of empirical research methods and the summary of frequent threats to validity motivates careful choice and use of research method. In Section 1.3, internal and external software quality is explained. Additionally, some example definitions of quality are presented in order to show how this term can represent a number of things depending on context. Having shown definitions of software quality from both the internal and external perspective, the concept of *measuring* software quality is treated in Section 1.4. Some structural properties of object-oriented (OO) software frequently employed in efforts to build models for estimating quality are described in

Section 1.5. The structural properties used in these models are often based on static analysis of source code. The shortcomings of static coupling analysis are debated in Section 1.6, followed by an introduction to the twelve dynamic coupling measures which are the focus of this thesis.

The latter part of this chapter describes the research into how dynamic coupling measures work. In Section 1.7 we state the objectives of our research, and elaborate on the methodology used to achieve them. Next, in Section 1.8, we explain how attaining these goals contribute to furthering the understanding of the measures. Related and possible future work is outlined in Sections 1.9 and 1.10, respectively.

1.1 Software Engineering

The late '60s saw the inception of what was then called the "software crisis". Previously, hardware constraints had limited development of software to an extent where large-scale projects were nearly impossible. As these limitations were gradually set aside, a crisis emerged. The problem was that large projects often had trouble completing their work, using more time and resources than anticipated. The cause of these difficulties was thought to be that many techniques used in small-scale development could not be scaled up to meet the needs of larger projects. The solution, many felt, was to apply principles commonly found in more traditional engineering disciplines to software development. Hence, the term "software engineering" was coined during the famous 1968 NATO conference in Garmisch, Germany [NR68].

In general, the application of engineering principles can be taken to mean an attempt at structured and gradual improvement of theories, methods, processes and tools (e.g., [PWC95]). Since the 1968 NATO conference, many new software engineering methods have been proposed. Unfortunately, they frequently have little or no properly documented success.

1.1.1 Empirical Software Engineering

The ideal way of establishing new approaches to software engineering is through research and practical small-scale application [FPG94]. This motivates the term *empirical* software engineering, in which software engineering methods are evaluated through empirical methods such as controlled experiments, surveys and case studies.

Fenton *et al.* [FPG94] describe how the software engineering industry is fraught with unsubstantiated claims of improvement. The authors refer to vendors advertising 250% productivity gain and maintenance effort reduced by 80%. Any rational development organisation will seek evidence of such claims to efficacy before applying new methods or new technology.

Per definition, research should be based on empirical findings, and not on hearsay or anecdotal evidence. Using empirical techniques can often help

researchers substantiate their results, increasing the likelihood that they are taken seriously by industry representatives and by each other.

As a research discipline, software engineering often has to deal with problems more commonly associated with research in psychology, medicine and the social sciences. The common denominator in these fields is that they involve people, and the fact that quantifying people's perceptions and reactions is difficult. Using an empirical approach is widely believed to be the best way of dealing with ambiguities often encountered in such settings.

1.2 Research Methods

There are a number of ways to conduct research in software engineering. For example, Adrion proposes categorising research based on the method used [Adr93]:

- **The scientific method** – Observe the world, propose a model or theory of behaviour, measure and analyse, validate hypothesis of the model or theory, and if possible: repeat.
- **The engineering method** (evolutionary paradigm) – Observe existing solutions, propose better solutions, build or develop, measure and analyse, repeat until no further improvements are possible.
- **The empirical method** (revolutionary paradigm) – Propose a model, develop statistical or other methods, apply to case studies, measure and analyse, validate the model, repeat.
- **The analytical method** – Propose a formal theory or set of axioms, develop a theory, derive results, and if possible compare with empirical observations.

The scientific, analytical and engineering approaches can be difficult to use in practical software engineering. The engineering method relies on the improvement of existing solutions. In a recent research field such as software engineering there are often no solutions to improve, partly due to the rapid pace of the industry and partly because of incomplete or unsubstantiated previous research [FPG94].

Both the scientific and the analytical method are based on subjecting a proposed model or theory to validation. This can sometimes be problematic in software engineering research, for a number of reasons. First, researchers often lack sufficiently accurate theories and models on which to base their investigations. This might in part be due to a lack of previous studies and material. Furthermore; forming theories and building models which correspond to a phenomenon's behaviour require understanding of the basic principles involved. This is often difficult as software engineering is closely tied

to how humans understand and represent abstract concepts [Cas02]. With software engineering itself relying on human cognition to such an extent, it is no wonder that research in the field is difficult [Her99]. Such research is founded, in essence, on abstract models and theories meant to represent the abstract process of creating software. "Abstracting an abstraction" might be one way to describe it.

The empirical approach has more room for exploratory analysis, and does not require the same amount of up-front theories and axiomatic models as the other three methods. It is therefore often better suited to conducting theory-creating research [Jar01]. Furthermore, if statistical prescriptions are followed [KPP+02], and if data is made available, this method lends itself well to replication.

1.2.1 Empirical Methods

Jarvinen [Jar01] divides empirical studies into two categories:

- **Theory-testing** – Attempt to determine if a hypothesis can be confirmed or falsified based on data from case studies, experiments or surveys. A prerequisite for theory-testing studies is some knowledge or data.
- **Theory-creating** – Seeks a theory that can be used to explain observations. This approach is suitable if there is no prior knowledge related to a phenomenon.

There are several methods to choose from in empirical software engineering. The suitability of the various methods is ultimately determined by such things as the availability of existing theories, resources and the purpose of the research itself. The following list describes some common approaches to research in empirical software engineering:

- **Controlled experiments** – Controlled experiments are often used to evaluate relationships between phenomena found during initial exploratory case studies [And03]. This design isolates a phenomenon and controls conditions in an artificial setting. Usually, this can only be accomplished on a limited scale, which is why Kitchenham calls controlled experiments "research in the small" [KPP95]. While it is possible to obtain very specific information by using controlled experiments, they often suffer from problems related to realism [SAA+02]. As a consequence, results are not necessarily applicable outside the experimental setting [And03, KPP95].

  A controlled experiment begins by defining research goals, and the formulation of a hypothesis based on them. It proceeds by the development of needed material, such as forms and examples, setting up

required tools, and so on. The treatments prescribed in the experimental design should be applied to two or more groups of subjects, followed by statistical analysis of the obtained data. Evaluation of the results, possibly in light of previous knowledge, should yield a conclusion.

- **Surveys** – Surveys can be employed when aiming for an overview or the state of practise on a large scale. Kitchenham [KPP95] call surveys "research in the large". This empirical research method combines some of the advantages of formal experiments with those of case studies. Because surveys are conducted on a large scale the problems often associated with replication of case studies are avoided (as in controlled experiments). Furthermore; as surveys are not conducted in an experimental setting their results are often directly applicable to real-world scenarios (as in case studies). However, according to Kitchenham [Kit96a], surveys should only be used to demonstrate association, not causality.

  Empirical surveys are usually conducted by posing structured predefined questions to a population sample [Jar01]. One problem related to this method is that there might be discrepancies between how researcher and research subject interpret the meaning of various questions and answers [Ari01].

- **Case studies** – Case studies provide researchers with the possibility of evaluation in a real-world setting without the scientific rigour of experiments and surveys. While findings in controlled experiments are sometimes difficult to apply outside the context of the research setting, results from case studies are often directly applicable. However, it is sometimes difficult to interpret such results, and they can not necessarily be applied in settings or environments outside the context of the study.

  Kitchenham call case studies "research in the typical" and proposes guidelines for how they should be conducted and evaluated [KPP95].

1.2.2 Criticism of Empirical Research in Software Engineering

Empirical software engineering has been criticised on a wide range of issues related both to conducting controlled experiments and application of statistical methods. Many objections to the current state of the art can be placed in the following categories:

- **Lack of empirical verification** – If research in software engineering is to have an impact within the industry it is vital that theories are backed

by solid empirical work. According to Fenton [FPG94], far too much published material can be described as "analytical advocacy research", proposing theories and deriving potential analytically, without conducting empirical investigations. Glass assumes a similar view [Gla94]. Zelkowitz and Wallace examined 619 papers published in the highly acclaimed IEEE Transactions on Software Engineering in 1985, 1990 and 1995 [ZW98]. They found no evidence of empirical validation in 36%, 29% and 19% of the papers, respectively.

- **Lack of realism in empirical studies** – Unfortunately, empirical studies in themselves do not guarantee correctness. Zelkowitz and Wallace state that *"All too often the experiment is a weak example favouring the proposed technology over alternatives. Sceptical scientists would have to view these experiments as potentially biased."* [ZW98].
  Fenton *et al.* [FPG94] distinguishes between "toy" and "real" studies, and argue that although "toy" studies (i.e., using student subjects) are less expensive their prevalence influence findings.
  Sjøberg *et al.* [SAA+02] defines the concept of *"mundane realism"* as the resemblance of the experimental condition to the real world, and state that this is a requirement in order to apply research results in an industrial setting. They define three prerequisites for such realism; realistic tasks, realistic subjects and realistic environments.
  Although many of these statements are found in papers related to conducting studies involving people, they are still relevant for the wider context of empirical studies in general.

- **Errors in use of statistical methods** – There is some concern over current application of statistical techniques in software engineering research. Kitchenham *et al.* make this clear, and refer to similar problems in medical journals as proof of their point [KPP+02]. They also propose guidelines for how statistical methods should be applied in six different areas; experimental context, experimental design, conducting experiments and data collection, analysis, presentation of results and interpretation of results.

- **Lack of replication** – Yet another matter of concern is the lack of replication in software engineering research. In [BDM+95] the authors state that *"without the confirming power of external replication, results in experimental software engineering should only be provisionally accepted, if at all"*. Miller notes three potential problems facing empirical software studies: *"low statistical power, large number of potential covariates with the treatment variable and verification of the process and products of the study."* [Mil]. His conclusion is that these limitations can be overcome by more frequent replication of studies.

- **Lack of operational definitions** – Some software measures require additional interpretation by the person responsible for applying them in practise. This introduces a possible source of misinterpretation, as people will inevitably understand the measures differently.
  An *operational* definition does not need additional interpretation in order to be used in practise [BDW98]. Pickard *et al.* argue that it is vital to *"define all software measures fully, including the entity, attribute, unit and counting rules"* [KPP+02]. Operational definitions are vital in preventing subjective, and thereby differing, interpretations of software measures[1].
  Briand *et al.* have criticised lack of operationality in relation to cohesion and coupling measures [BDW98, BDW99]. Kitchenham makes similar observations regarding the operationality of the ISO software quality model (ISO-9126, from 1992) [Kit96b].

1.3 Software Quality

Software quality is an intangible attribute in itself, as it can be a number of different things depending on context. A software product might be perceived as "low quality" from, for example, a maintenance perspective, while at the same time being of "high quality" from the perspective of users. The definitions of software quality given by Boehm *et al.* [BBL76] and the ISO/IEC 9126-1 [IC01] represent compartmentalised notions of quality. Each compartment and its associated sub-characteristics describe a fairly coherent aspect of quality. For example, in ISO/IEC 9126-1 [IC01] "reliability" depends on "maturity", "fault tolerance" and "recoverability".

1.3.1 External and Internal Software Quality

Software quality attributes can be divided into two groups; external and internal attributes. The former are external in the sense that they are most easily observed after software has been put into use [Som98, p. 624]. For example, ISO/IEC 9126-1 states that "reliability" can be measured externally by recording the number of failures during a specified period of execution [IC01].

In contrast, *internal* quality attributes can be measured directly from the software product itself. These attributes are more closely related to the structural properties of a software product than the external attributes, and can to some extent capture and quantify information about the design of a program.

[1] Systems of related measures that facilitate quantification of some particular characteristic are also commonly referred to as "metrics" in software engineering literature.

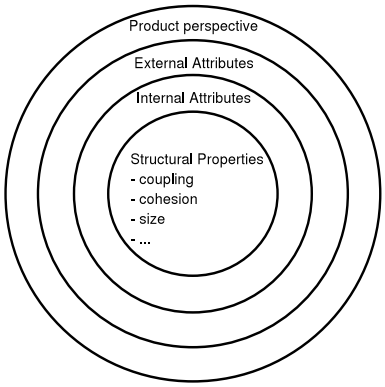

Figure 1.1: Relationship between external/internal attributes and structural properties of software

In the context of ISO/IEC 9126-1 [IC01], "reliability" is called an external quality indicator, while its sub-characteristics are referred to as internal indicators. Kitchenham and Sommerville do not include internal quality attributes in their models, as their emphasis is on explanation rather than operationality.

It is assumed that there exists a relation between specific *external* and *internal* quality attributes (see Figure 1.1). ISO/IEC 9126-1 [IC01, pp. 14] specifies that: *"The internal attributes are said to be indicators of the external attributes"*. It should therefore be possible to build empirical models which can be used to assess external quality based on internal attributes.

Once established, such models can be employed to predict the external quality of software systems before they are put into use. This might enable developers to focus design and development efforts in a manner likely to better the external aspects of software quality related to structure and design [BW02]. For example, if "reliability" is found to be influenced by one or more internal attributes, one might be able to predict possible rates of failure early in the development cycle. If the predicted values are excessive, steps can be taken to remedy the situation. This can possibly result in a lower rate of failure once the software product is put into use.

What remains is the question of exactly how it is possible to measure the internal quality attributes themselves. One method of obtaining such estimates is by examining the structural properties of the software product. Figure 1.1 outlines the relationship between external/internal quality attributes and structural properties. However, there are a number of different structural properties to consider, and determining which ones are suitable often requires a number of empirical studies.

Kitchenham lists three conditions which must be satisfied if an internal attribute is to be considered a useful predictor of external quality [Kit90].

1. The internal attribute must be measured accurately.

2. A relationship must exist between what we can measure and the external behavioural attribute.

3. This relationship must be understood, and it must be possible to express it in terms of a formula or model.

1.3.2 Example Definitions of Software Quality

Commonly seen definitions of quality range from fairly abstract to more technical, depending on context. Some authors view software predominately from an engineering perspective, and focus on the product itself, while others tend to view the software product as a part of the organisation which creates and maintains it.

It is possible to divide the different definitions of software quality into two categories based on their level of detail. Some definitions, like those given by Kitchenham and Sommerville [Som98, Kit96b], are quite abstract. In contrast, the definitions given by Boehm *et al.* [BBL76], and by ISO/IEC 9126-1 [IC01] go further in defining the various sub-characteristics which constitute the more abstract definitions of quality.

Kitchenham

Kitchenham divides software quality into three different perspectives [Kit96b].

- **User perspective** – The extent to which the software product meets the needs and requirements of users.

- **Manufacturing perspective** – Associated with the cost of maintenance, extendability and other properties of software in the context of users, organisations and developers.

- **Product perspective** – Characteristics of the software product itself (e.g., internal structural properties).

Sommerville

Sommerville distinguishes between four different quality perspectives in [Som98, p. 6]. Each perspective holds different sub-characteristics, although Sommerville does not state them explicitly. Sommerville's definitions of software product quality attributes are somewhat similar to those used by Kitchenham [Kit96b].

- **Maintainability** – It should be possible to change software to meet new requirements.

- **Dependability** – Software should not cause economic or physical damage.
- **Efficiency** – Good software should not waste system resources.
- **Usability** – It is important for software to have an appropriate user interface and documentation.

Boehm *et al.*

In a classic paper on quantitative assessment of software quality, Boehm assembles an hierarchical structure of quality characteristics [BBL76]. The structure contains four layers of detail. The third layer is interesting as there are parallels to it in Sommerville's work. The following list contains the characteristics at level three of the hierarchy, followed by the attributes they depend on (from level four). Some level three characteristics depend on the same level four attributes.

1. **Portability** – Device independence, self-containedness
2. **Reliability** – Self-containedness, accuracy, completeness, robustness/integrity, consistency
3. **Efficiency** – Accountability, device efficiency, accessibility
4. **Human engineering** - Robustness/integrity, accessibility, communicativeness
5. **Testability** – Accountability, accessibility, communicativeness, self-descriptiveness, structuredness
6. **Understandability** – Consistency, self-descriptiveness, structuredness, conciseness, legibility
7. **Modifiability** – Structuredness, augumentability

Item 5 to 7 are prerequisites for the characteristic "maintainability".

ISO/IEC 9126-1

One of the more recent additions to the body of quality definitions is the new ISO/IEC 9126-1 [IC01]. Published in 2001, it replaced the older ISO 9126 from 1992. The hierarchy of quality attributes presented in the ISO standard resembles the definitions used by Boehm *et al.* [BBL76].

The ordering of the attributes used in the ISO standard has been changed to highlight the similarity to Boehm's hierarchical definition of quality. Boldface text represents external quality attributes. Each external attribute depends on the internal quality attributes which follow it.

1. **Portability** – Adaptability, installability, co-existence and replaceability
2. **Reliability** – Maturity, fault tolerance and recoverability
3. **Efficiency** – Time behaviour and resource utilisation
4. **Usability** – Understandability, learnability, operability and attractiveness
5. **Functionality** – Suitability, accuracy, interoperability and security
6. **Maintainability** – Analysability, changeability, stability and testability

1.3.3 Comparison of Definitions

The difference between the definitions of software quality presented by Kitchenham and Sommerville manifests itself in that Kitchenham's definitions do not include anything specific about efficiency. Sommerville's focus seems to be more on the engineering aspect, describing attributes related to the software product itself. Kitchenham separates the product and its attributes from the process used to create it, and from the user perspective.

The definitions of quality given by Boehm *et al.* and in ISO 9126-1 differ from the other two definitions in their level of detail. This is probably because the definitions in both works are meant to be applied in practise. In contrast, Kitchenham and Sommerville do not concern themselves with practical application. Instead, they focus on presenting readers with an overview of different aspects of quality.

Both Boehm and the ISO model present sub-characteristics perceived to be related to the various external aspects of software quality. Their aim is that the various quality characteristics should be as orthogonal as possible. Boehm argues that it is therefore often pointless to combine different characteristics into overall measures of quality [BBL76], as such measures would not properly account for the various sub-characteristics. For example, highly portable and reliable software with low usability might be rated as having overall "good quality". Fenton has later used similar arguments [Fen94].

Kitchenham [Kit96b] and Arisholm [Ari01] criticise the choice of dependent characteristics in the ISO model, and point out that the choice of sub-characteristics sometimes seems arbitrary. For example, Kitchenham asks why portability is a top-level characteristic while interoperability is a sub-characteristic of functionality.

As the definitions of quality given by Boehm/ISO 9126-1 are meant to be usable in efforts to determine software quality, it would seem likely that the definitions they present are operationally defined. However, this is not the case [Kit96b]. Instead, the ISO/IEC 9126-1 recommends that the attributes

are measured directly on a software product, but gives no indication of how this should be done. It only suggests that if an attribute can not be measured directly, a related attribute should be measured instead. The ISO standard does not provide operational definitions, and does not follow the guidelines put forth in [KPP+02]. This makes practical application of the standard problematic.

ISO/IEC 9126-2 and 9126-3: External and Internal Metrics

While critique of lacking operational definitions was justified in relation to the old version of ISO 9126, this is not necessarily still the case. Since 1992 the ISO 9126 has been revised and is now called ISO 9126-1 [IC01]. In addition it has been augmented by two new standards. ISO/IEC 9126-2 "External metrics" [IC03a] and ISO/IEC 9126-3 "Internal metrics" [IC03b]. The internal metrics are intended for measurement of the software product itself, while the external metrics are meant to measure the behaviour of a computer-based system that includes software.

The two new standards are mainly composed of measurement definitions related to the quality sub-characteristics presented in ISO/IEC 9126-1. Although the perspective of the measures in the standards are different (external and internal, respectively), they are meant to estimate the same attributes, and the names of the measures themselves are the same.

The ISO committee seems to have considered recent debates regarding the use of measurement theory [FPG94] in literature on empirical software engineering when defining the two new standards. Definitions of the various measures in [IC03a] and [IC03b] carefully state which scales are used (e.g., nominal/interval/absolute/ratio/ordinal). They also provide formulae for calculating measurement values and specify what "good" values are.

However, there still seems to be a number of problems related to the standards. The most evident problem is that calculating the individual measures still depends on the subjective opinion of the person applying them (i.e., they are not operational). For example, the formula for calculating the external quality measure "functional compliance" is:

> *X = 1 - A/B*
> *A = Number of functionality compliance items specified that have not been implemented during testing.*
> *B = Total number of functionality compliance items specified.*
>
> [IC03a]

The problem inherent in this definition is that ISO/IEC 9126-2 does not define what a "functionality compliance item" actually constitutes. Therefore, the previously mentioned critique of ISO 9126 (from 1992) made by Kitchenham [Kit96b] is still relevant.

The relationship between the various measures used to estimate individual sub-characteristics also seems problematic. There are, for example, four external and four internal measures for estimating "suitability". How these should be combined, and how estimates of multiple sub-characteristics might be made into an overall measure of a main characteristic like "functionality" seems to be an open question.

The standard only recommends that the relationship between external and internal metrics be as strong as possible, and goes on to state that it is often difficult to design a rigorous theoretical model in which this relationship is clear. It explains that a hypothetical model, such as the one proposed by the standards, may contain ambiguities. ISO 9126-1 concludes that resolving such ambiguities might require that the relationship between external and internal attributes be modelled statistically in the course of using the metrics.

1.4 Software Metrics

Sommerville divides metrics into two broad categories based on their use [Som98]. Control metrics are applied in conjunction with tasks related to management of software projects. Typical examples of such measures are elapsed time, work effort and error density. The other category is referred to as internal predictor metrics. Predictor metrics are employed to measure product attributes, and can be used to assess product quality.

Goodman defines software metrics as:

> *"the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products"*
>
> [Goo93]

The definition provided by Goodman coincides with Sommerville's division into two categories. Both authors see metrics as tools that can be applied both to process and product.

This thesis is primarily concerned with metrics related to software products. Control metrics, and metrics related to improving the software development process fall outside our scope.

Establishing the relationship between internal structural properties and external quality attributes can be used in models for early prediction of software quality. Metrics are commonly used to quantify these internal structural properties. The remainder of this section will examine various measures for assessing the structural properties of software, and their relation to software product quality.

1.4.1 Overview

Many early works on software measurement focus on Lines Of Code (LOC or KLOC for thousand lines of code) as a measure of productivity, complexity and quality. Productivity can be derived by examining LOC per programmer/month. Quality might be related to defects per KLOC, and complexity can maybe be inferred directly from KLOC.

In the mid-1970's researchers started to recognise what was perceived to be problems related to using KLOC as a surrogate measure for complexity, effort and productivity. The perceived problem was at least partially that the expressive power of various programming languages varied widely. As different high-level languages came into use it was difficult to compare results across projects [FN99].

This led to an increased interest in other methods of quantifying software properties. McCabe's cyclomatic complexity [McC76] was a new method of identifying complexity based on analysing a program's decision structure. Albrecht [Alb79] pioneered function points as a measure of size independent of programming language. Halstead defined what has since been known as the Halstead Software Science metrics in [Hal77], attempting to derive programming effort, estimated number of defects and program size from the number of total and unique operators and operands found in a program.

1.4.2 Software Complexity

Some internal attributes are concerned with determining aspects of software complexity. Complexity is perceived to be important because software is ultimately created and maintained by people. If software is complex it is likely that its creators and maintainers will have more difficulty understanding and predicting the effects of the work they do on the product. That complexity has a confounding effect on developers and maintainers is assumed in several articles on the subject of software measurements [BW02, BDW99, CK91]. This assumption might in part be traced back to a 1972 paper by Parnas on decomposing systems into modules in order to increase comprehensibility and flexibility [Par72]. However, Parnas does not present any empirical evidence to support the claim.

There are some problems related to quantifying complexity. First, it should be made clear that complexity as it relates to software can be any number of different things. It is not possible to find a real-valued combined measurement which does justice to such diverse attributes as maintainability, reliability and changeability at the same time. The problem, according to Fenton [Fen94], is that such a combined measure will eventually have to satisfy conflicting aims. Fenton's example is "quality" of people. If two people, a and b, are characterised by 1) physical strength and 2) intelligence, it is not possible to find a single measure M which satisfies both $M(a) > M(b)$

(when a is stronger than b) and $M(a) > M(b)$ (when a is more intelligent than b). This is because these two "attributes" of people are not in any way related.

Another problem related to quantifying a program's complexity is the question of *who* it is complex for. Perceptions of complexity vary widely and are highly subjective. Arisholm and Sjøberg conducted an experiment, using both student and professional subjects, in which they examined differences in maintenance effort for two different types of OO program design [AS03]. One of the designs was based on delegation of control, while the other had a centralised control structure. Delegated functionality is often referred to as good OO design, while centralised structures are in many cases accused of being reminiscent of traditional procedural programming styles. Arisholm and Sjøberg found that junior developers used more time performing maintenance tasks on the delegate design than they did on the centralised one. For senior developers the exact opposite was the case; they spent more time maintaining applications with a centralised design than they did on maintaining a delegate structure. This indicates the extent to which perceptions of complexity depend on individuals and their level of experience.

As a result of the observations made by Fenton and Arisholm *et al.*, we are not concerned with determining overall complexity. Instead we attempt to measure and explore the relationship between a software product's structural properties and a single internal characteristic.

1.5 Measures for Object-Oriented Software

One way of making the internal attributes of a software product into operational definitions is by establishing their relationship to the product's structural properties. Figure 1.1 shows how specific structural properties might be related to internal, and ultimately, external attributes in a product perspective. However, heeding the discussion of complexity in Section 1.4.2, it is not necessarily possible to establish a direct relationship between internal and external attributes, or even between internal attributes and the structural properties themselves.

Structural properties of OO software can take a number of different forms. Some structural properties constitute a common theme in many of the papers which propose techniques for assessment of software product quality:

- **Size** – Size is perhaps the most commonly used structural property. It has been employed in conjunction with both procedural and object-oriented programming languages. It is often measured in LOC or KLOC. Generally, it is thought that large programs are more difficult to understand and maintain. By referring to Kitchenham [Kit90], Sommerville [Som98] claims that size can often be used to detect anomalous components with as much success as other, more sophisticated,

measures.
There are problems related to using size as a measure. Fenton [Fen94] comments on what appears to be a problem in predicting quality based on size by referring to the works of Tian *et al.* and Weyuker [TZ92, Wey88]. Weyuker's fifth property holds that for any two program bodies P, Q and a measure M, $M(P) \leq M(P;Q)$ and $M(Q) \leq M(P;Q)$. This property is consistent with a view of complexity being related to size. However, it is not consistent with a view of complexity as being related to comprehensibility. The reason is that according to Tian *et al.* [TZ92], comprehensibility is sometimes increased as size increases. Thus, Fenton concludes that a "size" type complexity measure M can satisfy Weyuker's fifth property, while a "comprehensibility" type complexity measure M can not.

- **Cohesion** – In the context of software development, cohesion is often defined as the extent to which different elements of a module belong together. The exact scope of a module varies. In programming languages like Java and C++, a module can be a package, a namespace or even a class. Cohesion is the extent to which the various functions contained in a module contribute to *one* common task. Modules with high cohesion are hypothesised to be easier to develop, reuse and maintain. They are also believed to be less fault-prone [BDW98]. There exists some empirical evidence to support this claim [CPM85, CCA86]. The concept of cohesion appears to have been introduced in conjunction with structured development techniques (structured programming) by Stevens *et al.* [SMC74]. Some of the theories regarding cohesion may even be traced back to a classic paper from 1972 on decomposing systems into modules [Par72].
  Various researchers have proposed measures based on cohesion. Briand *et al.* present an overview of 13 cohesion measures and a framework for classifying them [BDW98].

- **Coupling** – In 1974, Stevens, Myers and Constantine introduced the concept of using the amount of association between modules in a system as an indicator of poor design [SMC74]. Their idea was that module interdependence makes it difficult to 1) understand and change a system, and 2) gives rise to "ripple effects" which propagate errors and changes in one module to others.
  According to Briand *et al.* [BDW99], these principles were migrated to the context of object-oriented design by Coad and Yourdon [CY91]. However, in relation to object-oriented systems, coupling seems to be more complex as there are several mechanisms which can influence it. Coupling can occur between classes, objects, methods and as an effect of inheritance. Assessing the strength of various types of coupling is

at best difficult.

Briand *et al.* describe a comparative study of 30 different coupling measures defined in literature on software metrics [BDW99]. The proposed framework goes some way towards providing common criteria which can be used to classify and evaluate different coupling measures.

Large size, high coupling and low cohesion are generally thought to be indicators of complexity.

While size-based measures are good indicators of quality in many instances, they suffer from problems in others. Briand and Wüst recommend using size-related properties in conjunction with measures of coupling and inheritance [BW02]. Another problem related to size-based measures is the widely varying expressive power of various programming languages (see Section 1.4.1). The effect is that experiences from studies based on size measures are difficult to generalise outside the context of a specific language.

Cohesion measures have also been reported as difficult to use in practise. Briand and Wüst give two reasons for this in [BW02, p. 152]: The current understanding of the attribute is weak, and it is difficult to measure through static analysis of source code.

Among these structural properties, coupling has seen more interest than cohesion- and size-based measures. Measures based on coupling indicate how an object-oriented system's individual components use and depend on each other. High levels of coupling are generally thought to indicate complex systems, and are often associated with poor external quality, for example, low reliability and high maintenance effort [BDW99].

Example Measures for Object-Oriented Software: The C&K Metrics

The early '90s saw widespread adoption of the object-oriented (OO) programming paradigm in both commercial and academic software engineering communities. Most measures created up to that point were focused on estimating properties of software based on procedural (non-OO) languages. In 1991, Chidamber and Kemerer (often referred to as C&K) published their initial paper on measures for OO systems [CK91], followed by a more comprehensive study of the same measures in 1994 [CK94]. In principle their work was founded on two notions related to that period's state of the art software measures:

1. Most measures at the time were designed for procedural languages and problem solving strategies, and were therefore unsuitable in the context of OO systems.

2. Software measures were often without a solid theoretical basis, sometimes lacking important mathematical properties, consequently failing to display what might be termed normal predictable behaviour.

As suggested by the names of the various C&K measures, they are specific to object-oriented systems.

- **WMC** – Weighted methods per class.
- **DIT** – Depth of inheritance tree.
- **NOC** – Number of children.
- **CBO** – Coupling between objects.
- **RFC** – Response for a class.
- **LCOM** – Lack of cohesion in methods.

Chidamber and Kemerer specify different "viewpoints" for each of these measures. These viewpoints seem to be based on arguments in favour of the measures, and on suggested actions in cases where excessive values are found. For example, in [CK94], LCOM has four viewpoints. The first two read:

1. Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
2. Lack of cohesion implies classes should probably be split into two or more subclasses.

Since their initial publication, the C&K measures have been debated and criticised on grounds ranging from an absence of operational definitions [BDW98, BDW99] to a lacking focus on measurement theoretical principles [HM96]. Kitchenham *et al.* mention how Chidamber and Kemerer use Weyuker's properties [Wey88] as an example of how new measures are sometimes justified based on disputed criteria [KPF95]. Morasca *et al.* later refuted this claim [MBB+97].

1.5.1 Static versus Dynamic Structural Measures

The most common method of obtaining coupling, cohesion or size measures is based on parsing an application's source code, rather than its actual run-time behaviour. Gathering measurement data from source code is called "static analysis" and the resulting measures are referred to as "static measures" [EE01].

For example, Chidamber and Kemerer's measure definitions [CK91, CK94] are based on analysing source code. Their proposed measures are, in other words, static. This method of obtaining data is used both in subsequent refinements of their work, and in a majority of the measurement frameworks which have been developed.

It is also possible to gather measurement data from a running program. This approach is usually called "dynamic analysis", and the resulting measures are called "dynamic measures". Dynamic measures are less frequently used than static measures. There are two possible reasons for this. First, gathering data from a running application is usually more difficult than analysing relatively well-structured source code. Secondly, dynamic measures are less well-suited for use early in a development process when many software components are a long way from completion [EE01].

1.6 Dynamic Coupling Measures

Most coupling measures are static, i.e., based on analysing source code. However, because object-oriented applications often employ dynamic binding (polymorphism) it can be debated whether measurements obtained by parsing source code ("static analysis") give accurate results [Ari02].

Polymorphism will often make it seem like coupling occurs between classes high in an inheritance hierarchy, while actual coupling in fact takes place between descendants of abstract parent classes. The resulting measurements will often indicate that abstract classes, without any true functionality, are tightly coupled to a number of other classes. Examine Appendix C.1 for a example of how this occurs.

Using dynamic measures solves some of the problems related to static analysis, by obtaining data from a running program. The issues related to polymorphism become irrelevant, as it is always possible to determine exactly which class an object is instantiated from by inspecting it.

Unused or "dead code" occurs in almost all applications. It consists of classes and methods which are not in use, but that have not been removed from the source tree. Tools for obtaining static measures will read all the code presented to them, so unless the person performing the analysis knows the code base well and manually filters out unused code, the measurements can become inaccurate.

Dynamic analysis is based on the execution profile of an actual program. Because unused code is never executed, it does not become part of the data material. Dynamic coupling measures are therefore resistant to dead code.

The challenges posed by "dead" code and polymorphism lead Arisholm to develop dynamic coupling measures as a part of his doctoral thesis [Ari01]. He continued this effort in [Ari02].

1.6.1 Classification of Coupling Measures

The twelve coupling measures proposed by Arisholm [Ari01, Ari02] describe coupling according to the entity being measured, the coupling direction and the strength of the relation. These three elements distinguishes the measures from each other.

All of the measures have descriptive names. The format used is xC_xx, where C stands for *Coupling* and x represents wild-cards. Wild-cards can be any value allowed in that position of the measure name. For example, the first x indicates coupling direction; it can be either E (export) or I (import).

It is important to distinguish between different *directions* of measurement. The two entity types (described below) can both import and export functionality. Consequently, a distinction is made between *import* and *export* coupling. Import coupling represents method calls to external (non-local) entities, while export coupling represents method calls made to an entity by external classes or objects. Measures of import coupling are always named IC_xx, and measures of export coupling are called EC_xx.

In an OO system relying on inheritance it is possible to view interactions as occurring at either the class or the object level. The names of the various coupling measures indicate which *entity* is being measured. Object level measures are denoted by xC_Ox, while measures at the class level are denoted xC_Cx.

The final distinction between the measures is their *strength* criteria. Coupling strength is used to quantify the amount or "closeness" of association between entities. Strength can be seen as having three different levels of granularity. In more practical terms, the strength criteria indicate which types of entity associations are to be counted as coupling.

- **Distinct classes** – Accounts for interactions between distinct entities at the highest level. Exchange of one or more messages is counted as coupling. However, if two distinct entities exchange more than one message this does not increase the measured coupling. Denoted xC_xC.

- **Distinct methods** – At this level of granularity interactions between distinct methods of entities are counted. Multiple calls from/to the same entities/methods does not increase the counted coupling. Denoted xC_xM.

- **Dynamic messages** – This is the most fine-grained level of coupling measurement. Message uniqueness is based on source/target class/-method and the line number which the method call originates from. Denoted xC_xD.

A more exhaustive discussion of the various strength definitions and their implications when measuring coupling can be found in both Chapter 2 and Appendix B.8.

1.7 Research Objectives and Methodology

In this section we will define the objectives set for this research project. Additionally, we will report on the methodology used to attain them.

1.7.1 Formal Definitions

The first objective of this research was to:

- Formally define fully operational dynamic coupling measures.

Arisholm provides good textual descriptions and examples of how the various dynamic coupling measures are calculated [Ari01, Ari02]. However, formal definitions based on set theory and first order logic provide additional benefits. First of all they will, to some degree, ascertain the operationality of the dynamic coupling measures. Lack of operational definitions has been criticised by different authors (see Section 1.2.2). Furthermore, the availability of operational definitions will make replication of any empirical studies we undertake easier, as there will be less uncertainty about how the prescriptions for applying the measures should be interpreted.

Another good reason for formally defining the various coupling measures is that we are implementing a tool for collecting them. Creating such a tool is a time consuming task, and as anyone that has been involved in creating software knows, there are plenty of pitfalls in the transition from specification to product. The formal specification will aid us both in implementing the measures, and in verifying that the implementation complies with the specification.

1.7.2 Tool Support

The second objective of this research can be summarised as follows:

- Create a fully working tool for collecting the defined dynamic coupling measures.

One way of verifying to what degree the formal definitions really are operational is by implementing them. This will uncover most ambiguities and possibilities for misunderstanding inherent in the definitions of the measures.

Many research papers have been criticised for lacking empirical verification. In the case of software quality measures, such criticism is best avoided by implementing the proposed measures, and subjecting them to empirical validation in one or more studies.

The software engineering community have acknowledged that more frequent replication of empirical studies is desirable. A working implementation of the coupling measures enables independent parties to replicate our

research. Replication can be beneficial in several different ways: 1) It can help us establish with even greater certainty that the formal definitions of the coupling measures are really correct. 2) Replication may confirm that the implementation of the formal definitions are free of errors. 3) Because a working tool is available, any empirical studies we undertake can be repeated in order to establish our findings with greater certainty.

The coupling measures proposed by Arisholm [Ari01] were originally collected from a system written in SmallTalk. While SmallTalk is well-suited for collecting such measures because of its advanced run-time introspection features, it has one major drawback: It is not in widespread use, and is considered by many to be a rather academic programming language. This impacts the availability of candidate systems, and limits the usefulness of being able to collect data from SmallTalk.

Because of the drawbacks associated with using SmallTalk we have to consider collecting data from systems written in other OO programming languages. There are many possibilities, however, Java is an obvious choice for several reasons. It has been adopted by developers from business, academic and open-source communities. Consequently, there are many systems available for analysis. Furthermore, most Java implementations provide access to their internal APIs (application programming interfaces). Thus, as in SmallTalk, it is relatively easy to write tools which can access the internals of programs while they are executing.

The desire to obtain coupling measures from Java applications led to the development of Jdissect. Jdissect is a tool developed using C++ and its Standard Template Library (STL). The tool interfaces directly with the Java Virtual Machine (JVM), to collect data from a running Java application. This data can subsequently be analysed by Jdissect to obtain the coupling measures defined by Arisholm.

Implementation of the tool allows us to see how well the formal definitions work in practise, and possibly to uncover any ambiguities inherent in their definitions. It also provide means with which we can collect data for empirical validation of the measures.

1.7.3 Evaluation

The final objective of this project was to:

- Conduct a case study to investigate whether the proposed dynamic coupling measures provide data over and above what can be collected through use of static measures, by investigating their relation to change proneness and their practical usefulness as predictors of external quality.

It is quite common for different software product measures to be influenced by the same structural properties. Hence, it is important to determine

whether the various measures are really necessary, or if some of them actually capture the same information. If this is the case, it will be possible to eliminate the duplicate measures from future studies.

There are already numerous static coupling measures. In [BDW99] the authors compare 30 different such measures. Assessing whether our twelve dynamic coupling measures provide information in addition to static measures and class size is therefore an important goal.

As reported by Fenton, quality is not a tangible attribute which is easily summarised by any one single number [Fen94]. Consequently, we have to investigate just one aspect of quality, preferably one which can be quantified unambiguously. Change proneness was chosen after some deliberation, as this aspect of quality can be determined by examining two or more versions of a system.

In order to investigate the dynamic coupling measures and their relation to change-proneness it is necessary to determine which research method is best suited for the task. Clearly, a survey is not possible as there is little existing literature on the topic of dynamic coupling measures and their relation to change-proneness. Conducting a formal experiment is not ideal either, as we do not have enough data to properly formulate a theory. Furthermore, it is important that our results be applicable to situations outside the experimental context. The conclusion is that the evaluation of the dynamic coupling measures will have to take the form of a case study.

There are several important considerations to take into account when choosing candidate systems for this case study. Initially, there are two possibilities, 1) cooperate with a corporation or an institution in evaluating one of their existing Java systems, or 2) perform the evaluation of the measures on an open-source project. Cooperating with industry often introduces its own set of unique problems and constraints [Ari01]. It is often difficult to get companies to devote time and resources, and defining goals in which they too have vested interests.

While there are problems in relation to investigating open-source projects as well, they have some characteristics which make them very attractive. First, they usually keep source code revisions available to the public, a trait we deemed important as we were to investigate change-proneness. Additionally, some of these projects have been active for a long time, going back as far as 1998. This is positive, as there are a lot of revisions to examine. Second, there are a large number of projects to choose from. For example, there are over 22 Java tools, applications and frameworks organised under the umbrella of the Apache Jakarta project. Working with a company or corporation would possibly have meant a choice between only two or three applications.

Our case study examines Velocity, a template rewriting engine which has been actively maintained since March 2001. When we conducted our case study (May 2003) the project had released 17 versions. There was, in other

words, ample amounts of realistic data on which to base our study.

In performing this evaluation we indirectly address two frequent forms of criticism towards research in software engineering: 1) Research often lacks empirical verification, and 2) when empirical studies are in fact undertaken they often suffer from a lack of realism. The question of realism will not be settled by the fact that a case study is to be conducted, but rather by the exact nature of the system we choose to investigate. Velocity is a system in active use, and is therefore a good candidate with respect to realism.

1.8 Contribution

The overall contribution of this thesis can be summarised as follows.

Firstly, we provide formal and operational definitions of twelve dynamic coupling measures for object-oriented systems. These measures are meant to complement existing coupling measures based on static analysis. Because the twelve dynamic measures account precisely for inheritance, polymorphism, dynamic binding and "dead" code, we hope they will enable the design of better decision and prediction models.

Secondly, we describe a tool built to collect the proposed dynamic coupling measures. This goes some way towards ensuring the operationality of the measures and allows us to validate them in an empirical study. The tool might also enable independent parties to replicate our research.

Finally, our case study contributes to the understanding of the proposed dynamic coupling measures on three accounts. 1) The study shows that the information inherent in the measures is not redundant with respect to previously defined static measures. 2) The dynamic coupling measures capture information in addition to effects that can be attributed to size, e.g., the size of a program or class in an object-oriented system does not represent the same information as the new measures. 3) Our results indicate that the dynamic measures are good predictors of change proneness. Thus, prediction models based on these measures may be useful in focusing restructuring of re-engineering work on components in a program which seem likely to undergo future change.

1.9 Related Work

In this section we provide a brief overview of some research projects related to coupling measurement, tools and case studies. This is not meant to be a comprehensive list, but rather an overview of material which might be interesting in relation to our work.

1.9.1 Coupling Measures

The most famous works in the area of measuring structural properties of OO software are possibly the papers by Chidamber and Kemerer [CK91, CK94].

Briand *et al.* has collected a number of measures related to both coupling [BDW99] and cohesion [BDW98]. These articles propose common formal frameworks and definitions for explaining cohesion and coupling measures. In cases where the original measures are ambiguous or otherwise not operationally defined, Briand *et al.* attempt to find the most likely unambiguous and operational definition. These two papers are at once both excellent references and provide an overview of the field.

Yacoub *et al.* propose three different dynamic measures for object-oriented software in [YAR99]. Two of them are coupling measures, while the third is a measure of operational complexity based on ROOMcharts[2] and McCabe's cyclomatic complexity [McC76].

1.9.2 Similar Tools

It appears that much of the work in the field of dynamic program analysis is related to program understanding, reverse engineering and visualisation. There are however a number of tools for performing static analysis of source code.

Brooks and Buell [BB94] implemented a subset of the measures proposed by Chidamber and Kemerer in [CK91]. Their tool measured programs written in C++. The measures implemented at the class level were depth of inheritance tree (DIT), class coupling (CC), response for a class (RFC) and number of children (NOC). At the system level their tool supports calculating the number of class hierarchies. The authors conclude that if such tools are ever to be widely used in industrial settings gathering the measurement data should be as non-intrusive as possible.

Shimba is a tool for goal driven static/dynamic reverse engineering of Java systems created by Systä *et al.* [SKM01]. In [SYM00], the authors describe how they have implemented the full suite of measures proposed by Chidamber and Kemerer [CK91]. The authors show that the measures can be applied in the context of decorating dependency graphs of a system. They suggest that this will be useful in locating areas which should be considered for refactoring and re-engineering efforts.

Cahill, Hogan and Thomas describe a tool for collecting a wide range of measures from Java programs [CHT02]. Their tools performs static analysis of source code. What is interesting is that the system is founded on the notion of different measures as plug-ins. This supposedly makes the process

[2] Basically, UML state charts extended for use in real-time object modelling and simulation.

of implementing new measures much easier, as a complete set of scaffolding does not have to be developed by each independent researcher.

1.9.3 Other Case Studies

Munson and Hall studied the correlation between four different aspects of software complexity and test effectiveness [MH96]. They partitioned the complexity domain into static and dynamic aspects, arguing that dynamic measurements could better account for the extent of execution activity, thereby providing a better basis for estimation of test effectiveness.

Wilkie and Kitchenham investigate if one of the measures proposed by Chidamber and Kemerer, CBO (Coupling Between Object), can be used to predict ripple effects in C++ applications [WK00]. Their study is based on changes made to an application over 2 1/2 years. The authors conclude that CBO can be used to locate change-prone classes, but that the measure is too coarse to serve as a successful predictor of ripple effects.

In [YAR00], Yacoub *et al.* perform a case study of how the measures defined in [YAR99] can be used for early risk assessment at an architectural level. They claim that their approach yields four results; 1) it enables identification of architectural components which will require significant development resources, 2) it can be used to estimate risk based on the connections between components, 3) it can also quantify uncertainty in aggregated risk factors for system components and 4) it can be used to determine uncertainty in aggregated risk factors for connections in a system.

Briand *et al.* perform a very interesting study of fault-proneness models in [BWW02]. They investigate whether such models, based on design measurements from one system can be applied to another. Their conclusion is that using fault-proneness models in this manner is far from straightforward.

1.10 Future Work

In the course of a research project such as this it is only natural to become aware of the need for, or possibility of, related topics and natural extensions of the work. In a broad sense, these improvements and extensions can be split into three categories. The first mostly deals with extending the measure definitions. The second category deals with technical improvements to Jdissect, while the third deals with possibilities for other case studies.

1.10.1 Defining New Measures

Although Chapter 2 concludes that some of the proposed coupling measures are not completely orthogonal, neither with respect to each other nor to static measures, there are still combinations of coupling attributes which have not been investigated. The Jdissect framework can easily be adapted to measure

other coupling-related properties of Java applications. Alternative coupling measures can be related to class-level attributes or static methods, or simply use new message uniqueness definitions (strength criteria).

1.10.2 Tool Expansion

In our case study much time was spent examining source code in order to create Jdissect filters (see Section 3.3.3), to ensure the desired level of code coverage. High levels of coverage are not necessarily needed for the measures to accurately represent an application. It might be interesting to discover the level of coverage needed for our coupling measures to be meaningful. Every application is, of course, different. So attempting to find some universal level of coverage would be meaningless. However, more research into this could perhaps give some insight into the power inherent in data collected from, for example, tests covering 70-80% of an application's source code.

There are a number of other improvements which can be made to the Jdissect tool. However, many of these improvements are very technical in nature, and as such they are better explained in the context of the implementation and design of the Jdissect tool. The remainder of suggested improvements to the tool are therefore located in Section B.9.

1.10.3 Possible Case Studies

At present, Jdissect has only been used to analyse one real-world application. It is possible that analysis of other applications will produce different results. Furthermore, analysing other applications will give results which can be compared to the Velocity study. This might make it possible to 1) verify the results and 2) see if other applications show the same coupling measures to be orthogonal.

In the case study described in Chapter 2 Jdissect is used to determine class change-proneness. However, other uses can be found for the coupling measures. They might be useful in trying to establish complexity as it relates to ripple-effects, error-proneness or other aspects of software quality. It is also possible that the measures can be employed in identifying sections of code that should be candidates for refactoring efforts. The only way to determine if other uses can be found for the tool and the proposed measures is by conducting other case studies.

1.11 Thesis Overview

Chapter 1 (this chapter) of the thesis introduces empirical methods, software quality and the use of quality measures in the wider context of software engineering. It is meant to provide an introduction to the concepts treated in-depth in Chapter 2.

Table 1.1: Overview of chapters and their content

Chapter	**Focus**	**Contents**
1	Introduction	Provides the context for the two remaining chapters.
2	Theory and case study	Presents the theoretical foundation for the coupling measures and results from the case study of Velocity.
3	Tool and implementation	Contains a description of the software created in accordance with the specifications lain out in Chapter 2. It also provides a more detailed presentation of the methods used to collect data for the case study presented in Chapter 2.

Chapter 2 formally defines the dynamic coupling measures which are investigated. It also describes a case study examining the relation between change proneness and the proposed coupling measures. This chapter has previously been published as a Simula Technical report [ABF03]. A revised version of this work is due to appear in IEEE Transactions on Software Engineering [ABFar].

Chapter 3 gives a more detailed account of the tool implemented to collect the dynamic coupling measures. This chapter starts with an overview of how data can be collected from an executing Java application. Next, an account of the basic design of the tool is provided. Before collecting data for the case study described in Chapter 2 it was necessary to ensure the validity of the implementation. The steps taken to ensure validity are outlined in turn. Chapter 3 ends with a description of how data was collected from the Velocity application, and a look at the technical choices made in the design of the tool.

The somewhat unorthodox structure of this thesis is largely due to the inclusion of Simula Technical Report TR 2003-5 [ABF03], in its entirety, as Chapter 2. This was necessary because the material in the report is closely linked to the remainder of this thesis.

References

[ABF03] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. Technical report, Simula Research Laboratory, TR 2003-5/Carleton University, Canada, TR SCE-03-18, 2003.

[ABFar] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, To Appear.

[Adr93] W. R. Adrion. Research methodology in software engineering, summary of the Dagstühl workshop on future directions in software engineering. *ACM SIGSOFT Software Engineering Notes*, 18(1):35–48, January 1993.

[Alb79] A. J. Albrecht. Measuring application developement. pages 83–92, Monterey CA, 1979. Proceedings of IBM Application Developement joint SHARE/GUIDE Symposium.

[And03] B. Anda. *Empirical Studies of Contruction and Application of Use Case Models*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2003.

[Ari01] E. Arisholm. *Empirical Assessment of Changability in Object-Oriented Software*. PhD thesis, University of Oslo, Oslo, 2001.

[Ari02] E. Arisholm. Dynamic coupling measures for object-oriented software. In *proc. 8th IEEE Symposium on Software Metrics (METRICS'02)*, pages 33–42. IEEE Computer Society, 4-7 June 2002.

[AS03] E. Arisholm and D. I. K. Sjøberg. A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Technical Report TR 2003-6, Simula Research Laboratory, 6 2003.

[BB94] C. L. Brooks and C. G. Buell. A tool for automatically gathering object-oriented metrics. volume 2 of *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference*, pages 835–838. IEEE, NAECON, 23-27 May 1994.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. Proceedings of the Second International Conference on Software Engineering, pages pp. 592–605. IEEE, 1976.

[BDM+95] A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Technical Report EFoCS-17-95, ISERN-96-10, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1995.

[BDW98] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[BDW99] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan./Feb. 1999.

[BJ95] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Addison Wesley Longman, Reading, Massachusetts, U.S.A., 1995.

[BW02] L. C. Briand and J. K. Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 59:97–166, 2002.

[BWW02] L. C. Briand, M. L. Walcelio, and J. K. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, July 2002.

[Cas02] F. Castel. Theory, theory on the wall. *Communications of the ACM*, 45(12):25–26, December 2002.

[CCA86] D. N. Card, V. E. Church, and W. W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, 12(2):264–271, 1986.

[CHT02] J. Cahill, J. M. Hogan, and R. Thomas. The Java metrics reporter - an extensible tool for OO software analysis. Ninth Asia-Pacific Software Engineering Conference, pages 507–516. IEEE, 4-6 Dec. 2002.

[CK91] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming: Systems, Languages and Applications*, volume 26, pages 197–211. SIGPLAN Notices, Oct. 1991.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CPM85] D. N. Card, G. T. Page, and F. E. McGarry. Criteria for software modularization. IEEE Eighth International Conference on Software Engineering, pages 372–377. IEEE, 1985.

[CY91] P. Coad and E. Yourdon. *Object Oriented Design*. Prentice Hall, 1st edition, 1991.

[EE01] K. El-Emam. A methodology for validating software product metrics, 2001.

[Fen94] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[FN99] N. Fenton and M. Neil. Software metrics and risk. FESMA 99 - 2nd European Software Measurement Conference, October 1999.

[FPG94] N. Fenton, S. L. Pfleeger, and R. L. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(4):88–95, 1994.

[Gla94] R. L. Glass. The software-research crisis. *IEEE Software*, 11(6):42–47, Nov. 1994.

[Goo93] P. Goodman. *Practical Implementation of Software Metrics*. McGraw Hill, London, 1993.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, June 1977.

[Her99] J. D. Herbsleb. Metaphorical representation in collaborative software engineering. Proceedings of the international joint conference on Work activities coordination and collaboration, pages 117–126. ACM, February 1999.

[HM96] M. Hitz and B. Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.

[IC01] JTC 1-SC 7 ISO Commitee. ISO/IEC 9126-1:2001 - software engineering - product quality - part 1: Quality model. Technical report, ISO/IEC, 2001.

[IC03a] JTC 1/SC 7 ISO Commitee. ISO/IEC 9126-2:2003 - software engineering - product quality - part 2: External metrics. Technical report, ISO/IEC, 2003.

[IC03b] JTC 1/SC 7 ISO Commitee. ISO/IEC 9126-3:2003 - software engineering - product quality - part 3: Internal metrics. Technical report, ISO/IEC, 2003.

[Jar01] P. Jarvinen. *On Research Methods*. Tiedekirjakauppa TAJU, 2001.

[Kit90] B. A. Kitchenham. *Measuring Software Development*. Software Reliability Handbook. Elsevier Press, 1990.

[Kit96a] B. A. Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, January 1996.

[Kit96b] B. A. Kitchenham. *Software Metrics: Measurement for Software Process Improvement*. Blackwell, 1996.

[KPF95] B. A. Kitchenham, S. L. Pfeelger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[KPP95] B. A. Kitchenham, L. Pickard, and S.L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El-Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[MBB+97] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, and M. V. Zelkowitz. Comments on "Towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):187–188, 1997.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[MH96] J. C. Munson and G. A. Hall. Estimating test effectiveness with dynamic complexity measurement. *Empirical Software Engineering*, (1):279–305, 1996.

[Mil] J. Miller. Replicating software engineering experiments: A poisoned chalice or the holy grail. Draft.

[NR68] P. Naur and B. Randell, editors. *Software Engineering - Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1052–1058, December 1972.

[PWC95] M. C. Paulk, C. V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Carnegie Mellon University / Software Engineering Institute / Addison-Wesley, Reading Mass., 1995.

[SAA+02] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokác. Conducting realistic experiments in software engineering. Proceedings of the 2002 International Symposium on Empirical Software Engineering. IEEE, 2002.

[SG94] The Standish Group. The chaos report. Technical report, The Standish Group, 1994.

[SKM01] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software Practice & Experience*, (31):371–394, Feb. 2001.

[SMC74] W. Stevens, G. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Som98] I. Sommerville. *Software Engineering*. Addison-Wesley, 1998.

[SYM00] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000), Zurich, Switzerland*, March 2000.

[TZ92] J. Tian and M. V. Zelkowitz. A formal program complexity model and its application. *Journal of Systems Software*, 17:253–266, 1992.

[Wey88] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[WK00] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in C++ application software. *J. Syst. Softw.*, 52(2-3):157–164, 2000.

[YAR99] S. M. Yacoub, H. H. Ammar, and T. Robinson. Dynamic metrics for object-oriented designs. pages 60–61, 1999.

[YAR00] S. M. Yacoub, H. H. Ammar, and T. Robinson. A methodology for architectural-level risk assessment using dynamic metrics. In *proc. 11th International Symposium on Software Reliability Engineering*, pages 210–221, 2000.

[ZW98] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *Computing Practices*, pages 23–31, May 1998.

Chapter 2

Dynamic Coupling Measurement for Object-Oriented Software

Erik Arisholm[1], Lionel C. Briand[2] and Audun Føyen[1]

[1]Simula Research Laboratory
Lysaker, Norway
erika@simula.no; audunf@ifi.uio.no

[2]Software Quality Engineering Laboratory
Computer and Systems Engineering
Carleton University, Ottawa, Canada
briand@sce.carleton.ca

This chapter has previously been published as Simula Technical report TR-2003-5 and Carleton TR SCE-03-18 [ABF03]. A revision has been submitted for review to IEEE Transactions on Software Engineering [ABFar]

Abstract

The relationships between coupling and external quality factors of object-oriented software have been studied extensively for the past few years. For example, several studies have identified clear empirical relationships between class-level coupling and class fault-proneness. A common way to define and measure coupling is through structural properties and static code analysis. However, because of polymorphism, dynamic binding, and the common presence of unused ("dead") code in commercial software, the resulting coupling measures are imprecise as they do not perfectly reflect the actual coupling taking place among classes at run-time. For example, when using static analysis to measure coupling, it is difficult and sometimes impossible to determine what actual methods can be invoked from a client class if those methods are overridden in the subclasses of the server classes. Similarly, static analysis is not a fully appropriate tool to account for dead code. Coupling measurement has traditionally been performed using static code analysis,

because most of the existing work was done on non-object oriented code and because dynamic code analysis is more expensive and complex.

This paper describes how coupling can be defined and precisely measured on the basis of dynamic analysis or equivalent dynamic models of the system. We refer to this type of coupling as dynamic coupling. A first empirical evaluation of the proposed dynamic coupling measures is reported in which we study the relationship of these measures with the change proneness of classes. Data from maintenance releases of a large Java system are used for this purpose. Preliminary results suggest that some dynamic coupling measures are strong indicators of change proneness and that they complement existing coupling measures based on static analysis.

2.1 Introduction

In the context of object-oriented systems, research related to quality models has focused mainly on defining structural metrics (e.g., capturing class coupling) and investigating their relationships with external quality attributes (e.g., class fault-proneness) [BW02a]. The ultimate goal is to develop predictive models that may be used to support decision making, e.g., decide which classes should undergo more intensive verification and validation. Regardless of the structural attribute considered, most metrics have been so far defined and collected based on a static analysis of the design or code [AS00, BS98, BW02a, BDM97, BDW99, BeA95, CS00, CK94, CDK98, HHL90, LH93]. They have proven on many occasions to be accurate predictors of external quality attributes, such as fault-proneness [BW02a], ripple effects after changes [BWL99, CKK$^+$00], and changeability [Ari01, AS00, CKK$^+$00]. However, as reported by several authors [BW02a, CS00, CDK98, DBM$^+$96, DSWR02, HCN98], many of the systems that have been studied showed little inheritance and, as a result, limited use of polymorphism and dynamic binding.

As the use of object-oriented design and programming matures in the industry, we observe that inheritance and polymorphism are used more frequently to improve internal reuse in a system and facilitate maintenance, e.g., in open source projects, application frameworks, libraries. The problem is that the static, coupling measures that represent the core indicators of most reported quality models [BW02a], lose in precision as more intensive use of inheritance and dynamic binding takes place. This is expected to result in poorer predictive accuracy of the quality models that make use of static coupling measurement.

Let us take an example, as illustrated in Figure 2.1, to clarify the issue at hand. Due to inheritance, the class of the object sending or receiving a message may be different from the class implementing the corresponding method. For example, let object a be an instance of class A, which is inher-

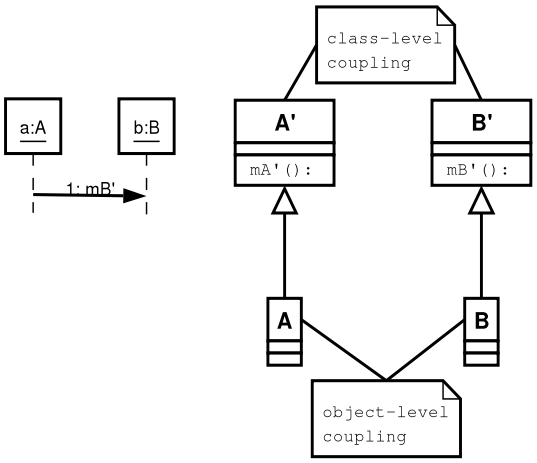

Figure 2.1: Class-level versus object-level coupling

ited from ancestor A' (Figure 2.1). Let A' implement the method mA'. Let object b be an instance of class B, which is inherited from ancestor B'. Let B' implement the method mB'. If object a sends the message mB' to object b, the message may have been sent from the method source mA' implemented in class A' and processed by a method target mB' implemented in class B'. Thus, in this example, message passing caused two types of coupling: (1) object-level coupling between class A and class B (i.e., coupling between instances of A and B), and class-level coupling between class A' and B'. The code may very well show statements where an object of type A invokes from mA' method mB' on an object of type B. However, to assume, through static code analysis, that there is class-level coupling between A and B as a result, is simply inaccurate. Both types of coupling, at the class and object levels, need to be captured accurately to address certain applications and must be investigated.

We propose here a set of coupling measures (referred to as dynamic coupling measures) that is defined on the basis of an analysis of what interactions are actually taking place between objects at run-time. They can be collected on the basis of a dynamic analysis of the code (Section 2.3.2) that is, by executing the code and saving information regarding the messages that are being sent among objects at run time. It is also, a priori, conceivable that dynamic design models (e.g., UML interaction diagrams [BRJ98]) could be used to collect such measures but, as discussed in Section 2.2.4, this presents a number of issues and practical challenges. Our objective in this report is therefore two-fold: (1) provide a precise definition of dynamic coupling measures and analyse their mathematical properties, (2) perform an empirical validation of the proposed measures by showing that dynamic coupling measures are useful indicators of a relevant quality attribute.

Even though dynamic coupling can be measured at the class level, just as static coupling can, it is important to note that they may result in significantly different measurements. For example, when the code shows that a method mA invoking, from an object of type A, a method mB on an object of type B, not only it may be the case (as in the example above) that the two methods are defined in ancestors of classes A and B, respectively, but they may actually be executed, at run time, on descendants of A and B. Static coupling inaccurately results in accounting such invocations as contributing to the coupling between classes A and B. On the other hand, dynamic coupling may result in class-level coupling between an ancestor of A and an ancestor of B, as well as object-level coupling between descendants of A and B, respectively.

Further evidence suggests that dynamic coupling could be of strong interest. For example, according to the results of a controlled experiment conducted in [ASJ01], static coupling measures may sometimes be inadequate when attempting to explain differences in changeability (e.g., change effort) for object-oriented designs. A related study indicates that the actual flow of messages is often traced systematically by professional developers when attempting to understand and change object-oriented software [BAJ01]. Furthermore, dynamic coupling is more precise than static coupling for systems with dead (unused) code. Coupling between classes that are never executed is not likely to be of high interest in most situations.

The remainder of this paper is organised as follows. Section 2.2 describes 12 dynamic coupling measures and highlights the ways in which they differ from static measures. These dynamic coupling measures differ in terms of the entity they measure and their scope and granularity, and are classified accordingly. They are defined in an informal, intuitive manner but also using a formal framework based on set theory and first-order logic. The main reason is to ensure that the definitions are precise and unambiguous to allow precise discussions of the measure properties and the replication of empirical studies. Section 2.3 presents a case study as a first empirical evaluation of the proposed dynamic coupling measures. Section 2.4 describes related research. Section 2.5 concludes and outlines future research.

2.2 Dynamic Coupling Measurement

We first distinguish different types of dynamic coupling measures. Then, based on this classification, we provide both informal and formal definitions, using a working example to illustrate the fundamental principles. Using a published axiomatic framework [BDW99], we then discuss the mathematical properties of the measures we propose. Our measures were designed so as to fulfil five properties that we deem very important for any coupling measure to be well formed. In order to define measures in a way that is programming

Table 2.1: Dynamic Coupling Classification

Entity	**Granularity (Aggregation level)**	**Scope (Include/Exclude)**
Object	Object Class (set of) Scenario(s) (set of) use case(s) System	Library objects Framework objects Exceptional use cases
Class	Class Inheritance Hierarchy (set of) subsystem(s) System	Library classes Framework classes

language independent, we refer to a generic data model defined with a UML class diagram.

2.2.1 Classifying Coupling Measures

There are different ways to define dynamic coupling, all of which can be justified, depending on the application context where such measures are to be used. First, since dynamic coupling is based on dynamic code analysis, coupling may be measured for a class or one of its instances. The entity of measurement may therefore be a class or an object.

Next, regardless of the entity of measurement, dynamic coupling measurement can be aggregated at different levels of granularity. With respect to dynamic object coupling, measurement can not only be performed at the object level, but also be aggregated at the class level, i.e., the dynamic coupling of all instances of a class is aggregated. In practise, even when measuring object coupling, the lowest entity of measurement is likely to be the class, as it is difficult to imagine how the coupling measurement of objects could be used. In a similar way, all the dynamic coupling of objects involved in a scenario[1] can be aggregated. We can also measure the object dynamic coupling in entire use cases (i.e., sets of scenarios), sets of use cases, or even an entire system (all objects of all use cases).

In the case where the entity of measurement is a class, the aggregation scale is different as we can aggregate dynamic class coupling across an inheritance hierarchy, a subsystem, a set of subsystems, or an entire system.

Another important source of variation in the way we can measure dynamic coupling is the scope of measurement. This determines which objects

[1]We use the term scenario here as it is used in a UML sequence diagram [6]: the sequence diagram models possible interactions in a use case and a particular path through the diagram represents a possible use case scenario.

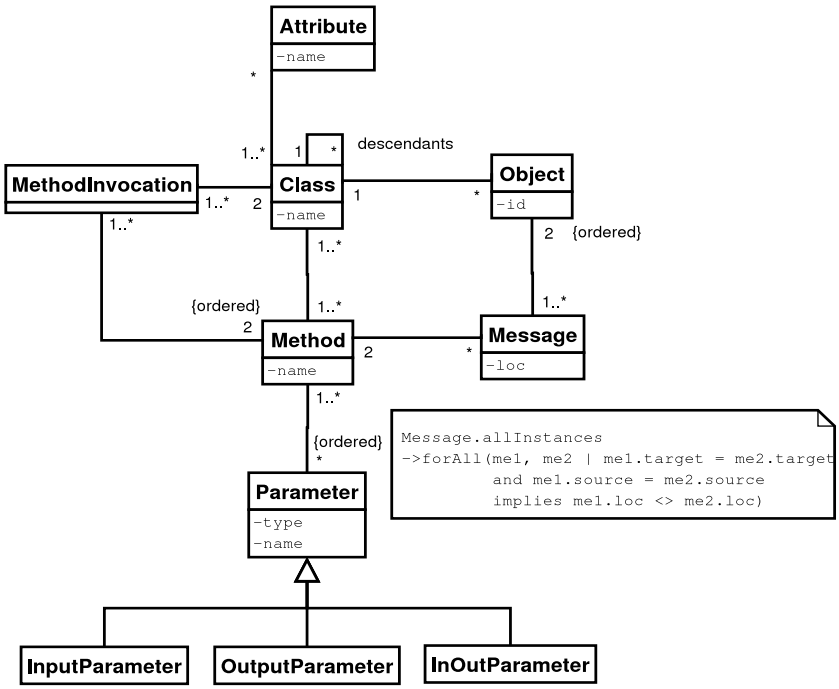

Figure 2.2: Class diagram capturing a data model of the dynamic analysis information

or classes, depending on the entity of measurement, are to be accounted for when measuring dynamic coupling. For example, we may not want, depending on the application context, to account for library and framework classes (e.g., classes from the SWING library in Java). At the object level, we may not want to account for certain use cases modelling exceptional cases (usually modelled as extended use cases [BRJ98]) or objects that are instances of library or framework classes. At the very least, we may want to distinguish the different types of coupling taking place in these different categories.

The choices we make regarding the entity, granularity, and scope of measurement depend on how we intend to apply dynamic coupling. Such choices form a classification of dynamic coupling measures that is summarised in Table 2.1.

2.2.2 Definitions

Before defining dynamic coupling measures, we introduce below the formal framework that will allow us to provide precise and unambiguous definitions. Not only do such definitions ensure that the reader understands the measures precisely, but they are also easily amenable to the analysis of their properties and facilitate the development of a dynamic analyser by providing precise

specifications. We provide a set of generic definitions that are based on the data model in Figure 2.2, which models the type of information to be collected. Each class and association in the class diagram corresponds to a set and a mathematical relation, respectively. The inheritance relationship corresponds to a set partition. Based on this, we define the measures using set theory and first order logic.

A few details of the class diagram in Figure 2.2 need to be discussed. Most role names are not shown, to avoid unnecessary cluttering of the class diagram. The meaning of associations is quite clear from the source and target classes. For example, methods are defined in a class, method invocations consist of a caller method in a source class and a callee method in a target class. Some of the key attributes are shown. One notable detail is that the line number where the target method is invoked is an attribute of a message that serves to uniquely identify it, as specified by the OCL [WK99] constraint shown in the class diagram. This is necessary, because the same target method may be invoked in different statements and control flow paths in the same source method. Message bearing those different invocations are considered distinct, because they are considered to provide different contexts of invocation for the method. As expected, method invocations between classes are differentiated from messages between objects.

Sets

The first step is to define the basic sets on which to build our definitions. These sets are derived from the data model in Figure 2.2.

- C: Set of classes in the system. C can be partitioned into the subsets of application classes (AC), library classes (LC), framework classes (FC). Some of these subsets may be empty, $C = AC \cup LC \cup FC$ and $AC \cap LC \cap FC = \varnothing$. Distinguishing such subsets may be important for defining the scope of measurement, as discussed above.
- O: Set of objects instantiated by the system while executing all scenarios of all use cases.
- M: Set of methods in the system (as identified by their signature).
- Lines of code are defined on the set of natural numbers ($\mathbb{N}$).

Relations

We now introduce mathematical relations on the sets that were defined above that will be fundamental to the definition of our measure.

- D and A are relations *onto* C ($\subseteq C \times C$). D is the set of descendant classes of a class and A is the set of ancestors of a class.

- ME is the set of possible messages in the system: $ME \subseteq O \times M \times N \times O \times M$. Indicated by the domain of ME, a message is described by a source object and method sending the message, a line of code ($\mathbb{N}$), and a target object and method.

- IV set of possible method invocations in the system: $IV \subseteq M \times C \times M \times C$. An invocation is characterised by the invoking class and method and the class and method being invoked.

- Other binary relations will be used in the text and their semantics can be easily derived from their domain and are denoted R_{Domain}. For example, $R_{MC} \subseteq M \times C$ refers to methods being defined in classes, a binary relation from the set of methods to the set of classes.

Consistency Rule

The relations IV and ME play a fundamental role in all our measures. In practise, an analysis of sequence diagrams or a dynamic analysis of the code allows us to construct ME. From that information, IV must be derived, but this is not trivial as polymorphism and dynamic binding tend to complicate the mapping. The consistency rule below specifies the dependencies between the two relations and can be used to develop algorithms that derive IV from ME.

$$\begin{aligned}(\exists(o_1,c_1),(o_2,c_2) \in R_{OC})(\exists l \in \mathbb{N})(o_1,m_1,l,o_2,m_2) \in ME \Rightarrow \\ (\exists c_1' \in A(c_1) \cup \{c_1\}, c_2' \in A(c_2) \cup \{c_2\}) \\ ((m_1,c_1') \in R_{MC} \wedge ((\forall c_1'' \in A(c_1) - \{c_1'\})(m_1,c_1'') \in R_{MC} \Rightarrow c_1'' \in A(c_1'))) \wedge \\ ((m_2,c_2') \in R_{MC} \wedge ((\forall c_2'' \in A(c_2) - \{c_2'\})(m_2,c_2'') \in R_{MC} \Rightarrow c_2'' \in A(c_2'))) \wedge \\ (m_1,c_1',m_2,c_2') \in IV \quad (2.1)\end{aligned}$$

Working Example

We now use a small working example, as shown in Figure 2.3, to illustrate the definitions above. Though it is assumed that our measures are collected through code static and dynamic analysis, we use UML to describe a fictitious example, because it is more legible than pseudocode. This example is designed to illustrate the subtleties arising from polymorphism and dynamic binding. Other aspects, such as method signatures, have been intentionally kept simple to focus on polymorphism and dynamic binding. The following sets can be derived from Figure 2.3:

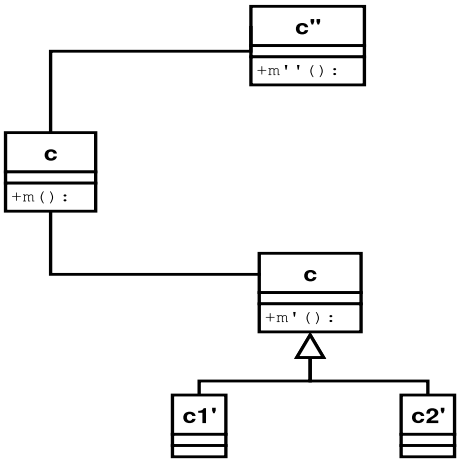

Figure 2.3: Working class diagram example (UML notation)

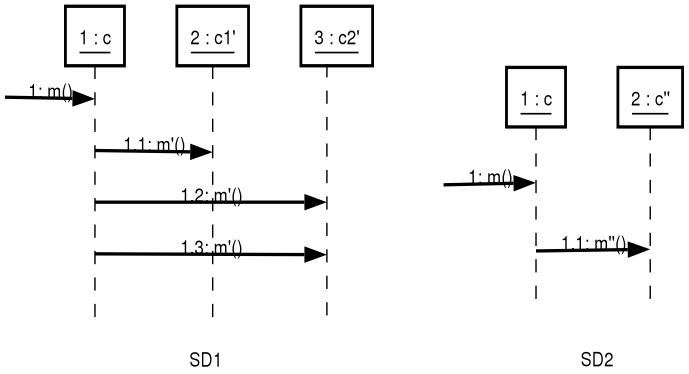

Figure 2.4: Two hypothetical sequence diagrams related to Figure 2.3

$$C = \{c, c', c'', c'_1, c'_2\}$$
$$M = \{m(), m'(), m''()\}$$
$$R_{MC} = \{(m, c), (m', c'), (m'', c'')\}$$

In order to derive other relevant sets and relations, let us introduce the sequence diagrams in Figure 2.4, where each message is numbered. Objects are referred to by using the sequence diagram number where they appear and their own identification number (i.e., $SD_i : ID$). O and R_{OC}, the relation that shows which instances are created for each class in the class diagram, can be derived from Figure 2.3:

$$O = \{SD_1 : 1, SD_1 : 2, SD_1 : 3, SD_2 : 1, SD_2 : 2\}$$
$$R_{OC} = \{(SD_1 : 1, c), (SD_1 : 2, c'_1), (SD_1 : 3, c'_2), (SD_2 : 1, c), (SD_2 : 2, c'')\}$$

Definitions of Measures

The measures are all defined as cardinalities of specific sets. Those sets are defined below and are given self-explanatory names, following the notation summarised in Table 2.2. First, as mentioned above, we differentiate the cases where the entity of measurement is the object or the class. Second, as in previous static coupling frameworks [BDM97, BDW99], we differentiate import from export coupling, that is the direction of coupling for a class or object. Furthermore, orthogonal to the entity of measurement and direction of coupling considered, there are at least three different ways in which the strength of coupling can be measured. First, we provide definitions for import and export coupling when the entity of measurement is the object and the granularity level is the class. Phrases outside and between parentheses capture the situations for import and export coupling, respectively.

- *Dynamic messages.* Within a run-time session, it is possible to count the total number of distinct messages sent from (received by) one object to (from) other objects, within the scope considered. That information is then aggregated for all the objects of each class. Two messages are considered to be the same if their source and target classes, the method invoked in the target class, and the statement from which it is invoked in the source class are the same. The latter condition reflects the fact that a different context of invocation is considered to imply a different message. In a UML sequence diagram, this would be represented as distinct messages with identical method invocations but different guard conditions.

- *Distinct method invocations.* A simpler alternative is to count the number of distinct methods invoked by each method in each object (that invoke methods in each object). That information is then aggregated for all the objects of each class.

- *Distinct classes.* It is also possible to only count the number of distinct server (client) classes a method in a given object uses (is used by). That information is then aggregated for all the objects of each class.

If we now look at where the calling and called methods are defined and implemented, the entity of measurement is the class and we can provide similar definitions. We then count the number of distinct messages originating

from (triggering the executions of) methods in the class, the number of distinct methods invoked by (that invoke) the class methods, and the number of distinct classes that the class is using methods from (that uses its methods).

Table 2.2 shows the formal set definitions of the measures when the granularity is the class, and the scope is the system. We provide an intuitive textual explanation only for the first set: $IC\_OM(c)$. Other sets can be interpreted in a similar manner.

$IC\_OM(c)$: A set containing all tuples (source method, source class, target method, target class) such that there exists an object o instantiating c (whose coupling is being measured) that sends a message to at least one instance of the target class in order to trigger the execution of the target method. The corresponding metric is simply the cardinality of this set. Note that the source class must be different from the target class ($c \neq c'$), because we are focusing on dependencies that contribute to coupling between classes, not their cohesion (as further discussed in [BDW98, BDW99]). Reflexive method invocations are therefore excluded.

Higher Granularities

If we want to measure dynamic coupling at higher levels of granularity, this can be easily defined by performing the union of the coupling sets of a set of classes or objects, depending on the entity of measurement. For example, if the entity of measurement is the class and the level of granularity is the subsystem, then to each subsystem SS there corresponds a subset of classes that it contains, $SC \in 2^C$, and we can define:

$$IC\_CM(SS) = \bigcup_{\forall c \in SC} IC\_CM(c)$$

To take a further example, when the entity of measurement is the object: To each use case UC there corresponds a set of participating objects $SO \in 2^O$ and we can define:

$$IC\_CM(UC) = \bigcup_{\forall o \in SO} IC\_CM(o)$$

Similar definitions can be provided for all levels of granularity.

Example

Returning to our working example in Figure 2.3 and Figure 2.4, we provide below all the non-empty coupling sets. Though, as a matter of convenience, our fictitious example is represented with UML diagrams, we refer to the

Table 2.2: Summary and acronyms for dynamic coupling measures (granularity=class, scope=system)

Direction	Entity of Measurement	Strength	Set Definition
Import **C**oupling	**O**bject	**D**ynamic Message	$IC\_OD(c) = \{(m, c, l, m', c')\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o, m, l, o', m') \in ME\}$
		Distinct **M**ethods	$IC\_OM(c) = \{(m, c, m', c')\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o, m, l, o', m') \in ME\}$
		Distinct **C**lasses	$IC\_OC(c) = \{(m, c, c')\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o, m, l, o', m') \in ME\}$
	Class	**D**ynamic Messages	$IC\_CD(c) = \{(m, c, l, m', c')\|$ $(\exists(o_1, c_1), (o_2, c_2) \in R_{OC})$ $(\exists l \in N)c \neq c' \wedge (o_1, m, l, o_2, m') \in ME \wedge$ $(\exists c \in A(c_1) \cup \{c_1\}, c' \in A(c_2) \cup \{c_2\})$ $((m, c) \in R_{MC} \wedge ((\forall c'_1 \in A(c_1) - \{c\})$ $(m, c'_1) \in R_{MC} \Rightarrow c'_1 \in A(c)))\wedge$ $((m', c') \in R_{MC} \wedge ((\forall c'_2 \in A(c_2) - \{c'\})$ $(m', c'_2) \in R_{MC} \Rightarrow c'_2 \in A(c')))\wedge$ $(m, c, m', c') \in IV)\}$
		Distinct **M**ethods	$IC\_CM(c) = \{(c, m, c', m')\|$ $(\exists(m, c), (m', c') \in R_{MC})$ $c \neq c' \wedge (m, c, m', c') \in IV\}$
		Distinct **C**lasses	$IC\_CC(c) = \{(m, c, c')\|$ $(\exists(m, c), (m', c') \in R_{MC})$ $c \neq c' \wedge (m, c, m', c') \in IV\}$
Export **C**oupling	**O**bject	**D**ynamic Messages	$EC\_OD(c) = \{(m', c', l, m, c)\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o', m', l, o, m) \in ME\}$
		Distinct **M**ethods	$EC\_OM(c) = \{(m', c', m, c)\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o', m', l, o, m) \in ME\}$
		Distinct **C**lasses	$EC\_OC(c) = \{(m', c', c)\|$ $(\forall(o, c) \in R_{OC}(\exists(o', c') \in R_{OC}, l \in N)$ $c \neq c' \wedge (o', m', o, m, l) \in ME\}$
	Class	**D**ynamic Messages	$EC\_CD(c) = \{(m', c', l, m, c)\|$ $(\exists(o_1, c_1), (o_2, c_2) \in R_{OC})$ $(\exists l \in N)c \neq c' \wedge (o_2, m', l, o_1, m) \in ME \wedge$ $(\exists c \in A(c_1) \cup \{c_1\}, c' \in A(c_2) \cup \{c_2\})$ $((m, c) \in R_{MC} \wedge ((\forall c'_1 \in A(c_1) - \{c\})$ $(m, c'_1) \in R_{MC} \Rightarrow c'_1 \in A(c)))\wedge$ $((m', c') \in R_{MC} \wedge ((\forall c'_2 \in A(c_2) - \{c'\})$ $(m', c'_2) \in R_{MC} \Rightarrow c'_2 \in A(c')))\wedge$ $(m', c', m, c) \in IV)\}$
		Distinct **M**ethods	$EC\_CM(c) = \{(m', c', m, c)\|$ $(\exists(m, c), (m', c') \in R_{MC})$ $c \neq c' \wedge (m', c', m, c) \in IV\}$
		Distinct **C**lasses	$EC\_CC(c) = \{(m', c', c)\|$ $(\exists(m, c), (m', c') \in R_{MC})$ $c \neq c' \wedge (m', c', m, c) \in IV\}$

line of code of the method invocation in message tuples, that we represent as $l(messageid)$. In the example, we assume that the line of code of the method invocations $m'()$ in messages $SD_1 : 1.1, SD_1 : 1.2$ and $SD_1 : 1.3$ are different. When the entity of measurement as well as the granularity is the class, we obtain the following import and export coupling sets:

$IC\_CD(c)$	$\{(m, c, l(SD_1 : 1.1), m', c'), (m, c, l(SD_1 : 1.2), m', c'),$ $(m, c, l(SD_1 : 1.3), m', c'), (m, c, l(SD_2 : 1.1), m'', c'')\}$
$IC\_CM(c)$	$\{(m, c, m', c'), (m, c, m'', c'')\}$
$IC\_CC(c)$	$\{(m, c, c'), (m, c, c'')\}$
$EC\_CD(c')$	$\{(m, c, l(SD_1 : 1.1), m', c'), (m, c, l(SD_1 : 1.2), m', c',),$ $(m, c, l(SD_1 : 1.3), m', c')\}$
$EC\_CM(c')$	$\{(m, c, m', c')\}$
$EC\_CC(c')$	$\{(m, c, c')\}$
$EC\_CD(c'')$	$\{(m, c, l(SD_2 : 1.1), m'', c'')\}$
$EC\_CM(c'')$	$\{(m, c, m'', c'')\}$
$EC\_CC(c'')$	$\{(m, c, c'')\}$

When the entity of measurement is the object, and the granularity is the class, we obtain the coupling sets below:

$IC\_OD(c)$	$\{(m, c, l(SD_2 : 1.1), m', c'_1), (m, c, l(SD2 : 1.2), m', c'_2),$ $(m, c, l(SD_2 : 1.3), m', c'_2), (m, c, l(SD_2 : 1.1), m'', c'')\}$
$IC\_OM(c)$	$\{(m, c, m', c'_1), (m, c, m', c'_2), (m, c, m'', c'')\}$
$IC\_OC(c)$	$\{(m, c, c'_1), (m, c, c'_2), (m, c, c'')\}$
$EC\_OD(c1')$	$\{(m, c, l(SD_1 : 1.1), m', c'_1)\}$
$EC\_OM(c1')$	$\{(m, c, m', c'_1)\}$
$EC\_OC(c1')$	$\{(m, c, c'_1)\}$
$EC\_OD(c2')$	$\{(m, c, l(SD_2 : 1.2), m', c'_2), (m, c, l(SD_2 : 1.3), m', c'_2)\}$
$EC\_OM(c2')$	$\{(m, c, m', c'_2)\}$
$EC\_OC(c2')$	$\{(m, c, c'_2)\}$
$EC\_OD(c'')$	$\{(m, c, l(SD_2 : 1.1), m'', c'')\}$
$EC\_OM(c'')$	$\{(m, c, m'', c'')\}$
$EC\_OC(c'')$	$\{(m, c, c'')\}$

The export coupling sets for c as well as the import coupling sets for c', c'', c'_1 and c'_2 are empty.

To gain a better insight into the impact of polymorphism on coupling, let us change the class diagram in Figure 2.3 by adding a new implementation of method $m'()$ in $c'_2 : R_{MC} = \{(m, c), (m', c'), (m', c'_2), (m'', c'')\}$, while keeping the sequence diagrams in Figure 2.4 unchanged. The new method implementation results in significantly changed import coupling sets for class c (removed elements are bold, whereas new elements are underlined):

$IC\_CD(c)$	$\{(m, c, l(SD_2 : 1.1), m', c'),$ $(\mathbf{m}, \mathbf{c}, \mathbf{l(SD_1 : 1.2)}, \mathbf{m'}, \mathbf{c'}), (\mathbf{m}, \mathbf{c}, \mathbf{l(SD_1 : 1.3)}, \mathbf{m'}, \mathbf{c'}),$ $\underline{(m, c, l(SD_1 : 1.2), m', c'_2)},$ $\underline{(m, c, l(SD_1 : 1.3), m', c'_2)}, (m, c, l(SD_2 : 1.1), m'', c'')\}$
$IC\_CM(c)$	$\{(m, c, m', c'), (m, c, m', c'_2), (m, c, m'', c'')\}$
$IC\_CC(c)$	$\{(m, c, c'), (m, c, c'_2), (m, c, c'')\}$

Adding a new implementation of an existing method in a subclass has resulted in increased import coupling for class c. This is because class c now imports from one additional class (c_2'), one additional method ($m'()$ in c_2'), and one additional distinct method invocation. However, object import coupling ($IC\_Ox(c)$) remains unchanged, as at the object level, instances of c were already importing from c_2'.

In a similar way, the export coupling of class c_2' has increased:

$EC\_CD(c')$	$\{(m, c, l(SD_1 : 1.1), m', c'), (\mathbf{m}, \mathbf{c}, \mathbf{l(SD_1 : 1.2)}, \mathbf{m'}, \mathbf{c'}), (\mathbf{m}, \mathbf{c}, \mathbf{l(SD_1 : 1.3)}, \mathbf{m'}, \mathbf{c'})\}$
$EC\_CM(c')$	$\{(m, c, m', c')\}$
$EC\_CC(c')$	$\{(m, c, c')\}$
$EC\_CD(c'')$	$\{(m, c, l(SD_2 : 1.1), m'', c'')\}$
$EC\_CM(c'')$	$\{(m, c, m'', c'')\}$
$EC\_CC(c'')$	$\{(m, c, c'')\}$
$EC\_CD(c2')$	$\{(m, c, l(SD_1 : 1.2), m', c_2'), (m, c, l(SD_1 : 1.3), m', c_2')\}$
$EC\_CM(c2')$	$\{(m, c, m', c_2')\}$
$EC\_CC(c2')$	$\{(m, c, c_2')\}$

2.2.3 Analysis of Properties

We show here that the five coupling properties presented in [BDW99] are valid for our dynamic coupling measures. The motivation is to perform an initial theoretical validation by demonstrating that our measures have intuitive properties that can be justified. We use IC_OM and IC_CM at the lowest granularity level (object, class) and system level as examples, but the demonstrations[2] below can be performed in a similar way for all coupling measures, at all levels of granularity.

Non-negativity

It is not possible for the dynamic coupling measures to be negative because they measure the cardinality of sets, e.g., IC_OM returns a set of pairs $(m, c) \in M \times C$.

Null values

At the system level, if S is the set that includes all the objects that participate in all the use cases of the system, $IC\_OM(S)$ is empty (and coupling equal to 0) if and only if the set of messages in S is empty:

$$ME = \varnothing \Leftrightarrow IC\_OM(S) = \varnothing$$

[2]These demonstrations are admittedly rather informal. We adopted a level of formality that we deemed sufficient to convince the reader these properties did indeed hold, without making the discussion unnecessarily terse.

This is consistent with our intuition as this should be the only case where we get a null coupling value. Since $ME = \varnothing \Leftrightarrow IV = \varnothing$ (consistency rule), we also have:

$$ME = \varnothing \Leftrightarrow IC\_CM(S) = \varnothing$$

At the object level, for $IC\_OM(o)$, we have:

$$(\forall o \in O, m \in M, l \in \mathbb{N}, o' \in O, m' \in M)(o, m, l, o', m') \notin ME \\ \Leftrightarrow IC\_OM(o) = \varnothing$$

Again, this is intuitive, as we should only obtain a null value if and only if object o does not participate to any message. Similarly, at the class level, we obtain:

$$(\forall o \in O, c \in C, (o, c) \in R_{oc}) IC\_OM(o) = \varnothing \Leftrightarrow IC\_CM(c) = \varnothing$$

(consistency rule)

Monotonicity

If a class c is modified such that at least one instance o sends/receives more messages, its import/export coupling can only increase or stay the same, for any of the coupling measures defined above.

If object $o \in O$ sends an additional message $(o, m, l, o', m') \in ME$, this cannot reduce the number of pairs $(method, class) \in R_{MC}$ that are part of the sets $IC\_OM(o)$ or $IC\_OM(S)$. The same can be said for export coupling if object $o \in O$ receives an additional message.

Adding a message to ME may or may not lead to a new method invocation in IV. But even if this is the case, the sets $IC\_CM(c)$ and $IC\_CM(S)$ cannot possibly lose any element.

Similar arguments can be provided for all coupling measures, at all levels of granularity. To conclude by adding messages and method invocations in a system, object and class coupling measures cannot decrease, respectively, thus complying with the monotonicity property.

Impact of merging classes

Assuming c' is the result of merging c_1 and c_2, thus transforming system S into S', for any *Coupling* measure, we want the following properties to hold at the class and system levels:

$$Coupling(c_1) + Coupling(c_2) \geq Coupling(c')$$
$$Coupling(S) \geq Coupling(S')$$

Taking IC_CD as an example, we can easily show this property holds: All instances of c_1 and c_2 in IV's tuples are substituted with C'. If there exist tuples of the type (m_1, c_1, m_2, c_2) in IV, then they are transformed into tuples of the form (m_1, c', m_2, c'). For IC_Cx measures, since we exclude reflexive method invocations because they do not contribute to coupling (Section 2.2.2), then tuples of the form (m_1, c', m_2, c') disappear because of the merging. Hence:

$$\left|IC\_CD(c')\right| \leq |IC\_CD(c_1)| + |IC\_CD(c_2)|$$

This property also holds for all other coupling measures.

Merging uncoupled classes

Following reasoning similar to that above, if two classes c_1, c_2 do not have any coupling, that means there is no tuple of the type (m_1, c_1, m_2, c_2) in IV. If we merge them into one class, we therefore cannot obtain tuples of the type (m_1, c', m_2, c'). Then, we can conclude IC_CD fulfils the following property:

$$\left|IC\_CD(c')\right| = |IC\_CD(c_1)| + |IC\_CD(c_2)|$$

This property also holds for all other coupling measures.

Symmetry between export and import coupling

By symmetry, for all class level dynamic coupling measures, we infer that the following property holds:

$$\bigcup_{\forall c \in C} EC\_Cx(c) = \bigcup_{\forall c \in C} IC\_Cx(c)$$

This stems from the fact that for any $(m, c, m', c') \in IV$, there is always a $l \in \mathbb{N}$ such that $(m, c, l, m', c') \in EC\_CD(c')$ and $(m, c, l, m', c') \in IC\_CD(c)$. Along the same lines, for each $(m, c, m', c') \in IC\_CM(c)$ and $(m, c, c') \in IC\_CC(c)$, there is a corresponding $(m, c, m', c') \in EC\_CM(c')$ and $(m, c, c') \in EC\_CC(c')$, respectively.

Following a similar argument when the entity of measurement is the object, we obtain:

$$\bigcup_{\forall o \in O} EC\_Ox(o) = \bigcup_{\forall o \in O} IC\_Ox(c)$$

The symmetry property is intuitive, because anything imported by a class or object has to be exported by another class or object, respectively. This condition applies at all levels of granularity.

Based on the property analysis above, we can see that our coupling measures seem to exhibit intuitive properties that would be expected when measuring coupling. This constitutes a theoretical validation of the measures. The next section focuses on their empirical validation, using project data.

2.2.4 Using UML Models for Data Collection

So far, we have assumed that dynamic coupling data are collected through dynamic analysis of the code. It was also suggested that using UML models presented a number of practical and technical challenges. However, measuring coupling on early design artifacts would be of practical importance because one could use that information for early decision making. For example, one could derive test cases and compute the dynamic coupling associated with each of the test cases based on UML diagrams. Test cases with high coupling could be exercised first, as they would be expected to uncover more faults and, therefore, the test plan would provide an order in which to run test cases based on dynamic coupling information.

The main problem lies with UML interaction diagrams. If we resort to UML diagrams for dynamic coupling measurement, we have to find a substitute for the line of code where the invocation is located to distinguish messages (in ME) and compute xx_xD measures. A natural substitute is the guard condition, which corresponds to different contexts of invocations.

An identical method on two messages with two distinct guard conditions must correspond to different invocation statements in the code. However, one guard condition on a message does not have to correspond to one invocation statement in the code. For example, one may have a guard of the form `[A or B]` that triggers the invocation of $m()$, and the corresponding code may show two distinct invocations statements for $m()$, each of them being in the body of an if statement with conditions A and B, respectively.

What this implies is that if xx_xD measures are collected from UML interaction diagrams, coupling will tend to be underestimated, because distinct elements of ME will not be distinguishable using UML interaction diagrams. However, the question is whether, in practise, this makes any significant difference. The advantages of using dynamic coupling measures on early UML artifacts may outweigh the drawbacks that are due to their lower precision. Furthermore, xx_xC and xx_xM measures are not affected by the use of UML interaction diagrams. If empirical investigation finds these

latter measures to be strongly correlated with xx_xD, it is doubtful the data collection inaccuracy discussed above will have any practical effect.

2.3 Case Study

This section presents the empirical results of a case study whose objectives are to provide a first empirical validation of the dynamic coupling measures presented above. The first subsection explains in more detail our objectives, the study settings, and the methodology we follow. In subsequent sections quantitative results are presented and interpreted.

2.3.1 Objectives and Methodology

To evaluate the dynamic coupling measures, an open-source software system called Velocity was used as a case study. Velocity is part of the Apache Jakarta Project [ASF04]. Velocity can be used to generate web pages, SQL, PostScript and other outputs from templates. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems. The system is implemented in Java and consists of more than 100 core application classes in addition to library classes. A total of 17 consecutive versions (versions 1.0b1 to version 1.3.1) of Velocity were available for analysis. The versions were released within a time span of approximately two years.

Several types of data were collected from the system. First, change data (i.e., using a class-level source code *diff*) was collected for each application class. Based on the change data, the amount of change (in SLOC added and deleted) of each class within a given set of consecutive versions was computed. Second, to collect the dynamic coupling measures, test cases provided with the Velocity source code was used to exercise each version of the system. Each test case was executed while a dynamic coupling tracer tool (Section 2.3.2) developed by the authors computed the dynamic coupling measures. Third, size and a comprehensive set of static coupling measures (defined in Appendix A.1 and A.2, respectively) were collected using a static code analysis tool. The scope of measurement was the application classes (AC) of Velocity. Thus, coupling to/from library and framework classes were not included (for further details, see Section 2.2.1).

One objective of the case study was to determine whether the dynamic coupling measures capture additional dimensions of coupling when compared with static coupling measures. Once this was verified, a subsequent objective was to obtain empirical evidence that dynamic coupling measures are indicators of external quality attributes and are complementary to existing static measures.

Following the methodology described in [BW02a], we first analysed the descriptive statistics of the dynamic coupling measures (Section 2.3.4). The

motivation was to determine whether they show enough variance and whether some of the properties we expected were visible in the data. The next step was to perform a principal component analysis (PAC) [Dun98], the goal of which was to identify what structural dimensions are captured by the dynamic coupling measures and whether these dimensions are at least partly distinct from static coupling measures. It is usual for software product measures to show strong correlations and for apparently different measures to capture similar structural properties. PAC also helps to interpret what measures actually capture and determine whether all measures are necessary for the purpose at hand. In our case, recall that we want to determine whether all xx_xC, xx_xM, and xx_xD measures are necessary, that is, to what extent they are redundant.

In order to investigate their usefulness as quality indicators, we investigate whether dynamic coupling measures are statistically related to change proneness, that is, the extent of change across the versions of the system we used as a case study. To do so, we analysed the changes (lines of code added and deleted) across classes of four subsequent sub-releases (called release candidates in Velocity) within one major release of the Velocity system (1.2). The dependent variable *(Change)* in this study is the total amount of change (source lines of code added and deleted) that has affected each of the 136 application classes participating in the test case executions across the 4 sub-releases of Velocity 1.2. Since none of these classes were added or deleted during the making of the successive releases, the variable *Change* is a measure of the change proneness of these classes, that is, of their tendency to change. Other possible dependent variables could have been selected, such as the number of changes, but we wanted our dependent variable to somehow reflect the extent of changes as well as their frequency. This assumes that there is a cause-effect relationship between coupling and change proneness, something which is intuitive because classes that strongly depend on or provide services to other classes are more likely to change, through ripple effects, as a result of changes in the system [BWL99]. Predicting the change proneness of a class can be used to aid design refactoring (e.g., removing "hot-spots"), choosing among design alternatives or assessing changeability decay [Ari01, AS00]. Change proneness has also been used in other studies as an indicator of maintenance effort [LH93].

One important issue is that not only do we want our measures to relate to change proneness in a statistically significant way, but we want the effect to be additional or complementary to that of static coupling measures and class size [BW02a, EEBGR01]. If some of the dynamic coupling measures remain statistically significant covariates when the static coupling measures and size measures are included as candidate covariates, this subset of dynamic coupling measures is deemed to significantly contribute to change proneness. We consider this to be empirical evidence of the causal effect between dynamic coupling and change proneness, of their practical usefulness, and hence we

consider it to provide an initial empirical validation of the dynamic coupling measures. More details are provided in Section 2.3.7.

2.3.2 Tool Support

It is a matter of some concern how to collect dynamic coupling data in a practical and efficient manner. A sophisticated tool was developed to collect the dynamic coupling data from Java programs. The tool separates the collection and analysis of dynamic coupling data into two phases. In the first phase, data from a running Java program is gathered and stored. This is accomplished by having the Java Virtual Machine (JVM) load a library of routines that are called whenever specified internal events occur. The interfaces used for communication between the JVM and the library are called JVMPI (Java VM Profiling Interface) and JVMDI (Java VM Debugging Interface). Most of the data is collected from the profiling interface. The JVMDI is used to obtain information about from which unique line number a method call originates (to obtain the information needed to calculate the xx_xD measures). During the data collection phase, the tool populates a data structure as specified in Figure 2.2. This data is then stored in a flat file structure.

In the second phase, the data is analysed. Another executable, sharing a great deal of code with the library, reads the flat files into a data structure identical to that used by the library. This structure is analysed to obtain the dynamic coupling measures. The analysis tool traverses the data structure in Figure 2.2 and computes the sets specified in Section 2.2.2. Each measure is then computed simply by counting the number of elements in each set. Raw data from several run-time sessions can be merged by the analysis tool, such that accumulated dynamic coupling data can be calculated. This merging capability enables the collection of coupling data for Java systems for which several concurrent instances of the JVM are used, such as large, distributed or component-based systems.

Our coupling tool uses the Java Virtual Machine to collect the message traces and other information specified in Figure 2.2. Another possible approach would be to instrument the source code to collect the needed run-time information. Such an approach would essentially modify the existing Java code to incorporate the data collection software within each application. Our approach provides several advantages over instrumentation. First, our tool does not require the Java source code to collect the dynamic coupling measures. This is a great advantage, for example, when analysing the coupling to library classes for which the source code may not be available. Another advantage is performance. Since our data collection tool is written in C++ and dynamically linked with the JVM at run-time, there is less performance overhead compared with an instrumentation approach in which the data collection software would have to be written in Java and then interpreted by the

Java VM along with the application source code. As performance overhead increases, the behaviour of concurrent software is more likely to be affected by the data collection process and it is important to minimise the chances of such a problem occurring.

2.3.3 Code Coverage

One practical drawback of using dynamic analysis is that one has to ensure that the code is sufficiently exercised to reflect in a complete manner the interactions that can take place between objects. To obtain accurate dynamic coupling data, the complete set of test cases provided with Velocity were used to exercise the system. Though this test suite was supposed to be complete, as it is used for regression test purposes, we used a code coverage tool and discovered that only about 70 percent of the methods were covered by the test cases. A closer inspection of the code revealed that a primary reason for this apparent low coverage was that a large number of classes were "dead" code. In addition, there were many occurrences of alternative constructors and error checking code that were never called. Fortunately, such code does not contribute to coupling. After removing the dead code and filtering out alternative constructors and error checking code, the test cases covered approximately 90 percent of the methods that might contribute to coupling among the application classes in Velocity. Consequently, the code coverage seems to be sufficient to obtain fairly accurate dynamic coupling measures for the 136 "live" application classes of Velocity 1.2.

2.3.4 Descriptive Statistics

This section discusses the descriptive statistics of the coupling and class size measures provided in Appendix A.3. These statistics are based on the first sub-release of the studied release (1.2) of Velocity. The first thing to notice is that the mean values for dynamic import coupling measures (e.g., IC_OC) are always equal to the mean values of their corresponding dynamic export coupling measure (e.g., EC_OC). This confirms the symmetry property discussed in Section 2.2.3. For most measures, there are large differences between the lower 25^{th} percentile, the median, and the 75^{th} percentile, thus showing strong variations in import and export coupling across classes. Many of the measures show a large standard deviation and mean values that are larger than the median values, with a distribution skewed towards larger values. Two of the static coupling measures show (almost) no variation and are not considered in the remainder of the analysis.

2.3.5 Principal Component Analysis

Principal Component Analysis (PCA) [Dun98] was used to analyse the covariance structure of the measures and determine the underlying dimensions

they capture. PCA usually generates a large number of Principal Components, which are usually retained or discarded based on the amount of variance they explain1. Appendix A.4 provides the results of PCA when accounting for dynamic coupling measures only. Appendix A.5 provides the results of PCA when considering all measures. When considering dynamic coupling measures in isolation it becomes obvious that all xx_xC, xx_xM, and xx_xD measures belong to identical components and capture similar properties. This implies that it may not be necessary to collect all of these measures, and in particular, the xx_xD measures that cannot be collected on UML diagrams and require dynamic code analysis. It is interesting to note that this confirms the results in an earlier case study on a Smalltalk system [Ari02].

In the PCA involving all measures, two principal components (PC5 and PC7) clearly capture export dynamic coupling and import dynamic coupling, mostly at the object level (i.e., object-level show higher weights), respectively. As for all PCA results when many measures are included, some of the principal components are difficult to interpret. The first one, for example, captures most size measures and some import static coupling measures, but also, to a lesser extent, import dynamic coupling at the class level. As has been observed in past studies [BW02b, BWIL99], size may be to some extent related to some of the coupling measures. With respect to dynamic coupling, results show that class-level measures are moderately correlated with some of the size and static coupling measures, but overall, the PCA analysis seems to indicate that our dynamic coupling measures (especially when the entity of measurement is the object) are not redundant with existing static coupling and size measures. The next sections go even further in this respect by providing evidence that dynamic coupling measures are also useful quality indicators.

2.3.6 Relationships between Change Proneness and Dynamic Coupling

The goal of this section is to evaluate the extent to which each of the dynamic coupling measures are related to our dependent variable, change proneness (see Section 2.3.1). However, since the size (SLOC) of a class is an obvious explanatory variable of Change (SLOC added + deleted), it may be more insightful to determine whether a coupling measure is related to change proneness independently of class size. We therefore tested whether the dynamic coupling measures are significant additional explanatory variables, over and above what has already been accounted for by size. The underlying assumptions are that the larger the export coupling, the more likely a class is to be changed, because it has to adjust to the evolving needs of many classes. Similarly, the larger the import coupling, the more likely a class is to be changed, because it depends on many other classes that may

Table 2.3: Relationships between change proneness and dynamic coupling

Regression Covariates	Coefficient Size	p-value Size	Coefficient Coupling	p-value Coupling	R-Sq	R-Sq (adj)
CS1	**0.068**	**0.000**	**N/A**	**N/A**	**12.8%**	**12.1%**
CS1, IC_OC	0.067	0.000	0.123	0.778	12.8%	11.5%
CS1, IC_OM	0.067	0.000	0.085	0.769	12.8%	11.5%
CS1, IC_OD	0.068	0.000	0.010	0.971	12.8%	11.5%
CS1, IC_CC	0.059	0.001	1.038	0.151	14.1%	12.8%
CS1, IC_CM	0.059	0.001	0.748	0.165	14.0%	12.7%
CS1, IC_CD	0.063	0.000	0.314	0.473	13.1%	11.8%
CS1, EC_OC	**0.064**	**0.000**	**1.656**	**0.001**	**20.1%**	**18.9%**
CS1, EC_OM	0.065	0.000	0.899	0.009	17.2%	16.0%
CS1, EC_OD	**0.065**	**0.000**	**0.830**	**0.002**	**19.0%**	**17.7%**
CS1, EC_CC	**0.061**	**0.000**	**1.758**	**0.000**	**20.6%**	**19.4%**
CS1, EC_CM	0.064	0.000	0.736	0.017	16.5%	15.2%
CS1, EC_CD	0.065	0.000	0.469	0.024	16.1%	14.8%

themselves change, thus triggering ripple effects.

To achieve this, we systematically performed a multiple linear regression involving class size (SLOC) and each of the dynamic coupling measures and then determined whether the regression coefficient for the coupling measure was statistically significant (using a standard statistical t-test [FW98]). This resulted in 12 coupling measures and one size measure being tested for significance and with that many tests, the discovery of empirical relationships by chance becomes more likely [CG93]. Consequently, the significance level (alpha-level) was set to $\alpha = 0.05/13 = 0.004$, following the Bonferroni procedure {Christensen, 1996 #152}. Regression coefficients were considered significant if the t-test p-value was smaller than 0.004. However, the Bonferroni procedure is conservative and the reader may choose to be less strict when interpreting the actual p-values in Table 2.3.

The results (Table 2.3) show strong support for the hypotheses that three of the dynamic export coupling measures are clearly related to change proneness, in addition to what can be explained by size in SLOC (CS1). On the other hand, dynamic import coupling measures do not seem to explain additional variation in change proneness, compared to size alone. Once again, this confirms the results obtained in an earlier case study on a Smalltalk system [Ari02].

The coefficients of determination (R-Sq) are not high, but that is to be expected, because we only include size and one coupling measure at a time and, as a result, a large portion of the variance is still not accounted for. A few observations had very large residuals that contributed to the low coefficients of determination and, thus, the underlying regression model assumption of normally distributed residuals is violated due to these outliers. Removing them significantly improved the model fit while still confirming the results of

the models in Table 2.3. This indicates that the model violations are of little practical consequence with regards to the results of the hypotheses tests. The following section evaluates the extent to which the dynamic coupling measures are useful predictors when building the best possible models by using size, static coupling, and dynamic coupling measures as possible model covariates.

2.3.7 Prediction Model of Change Proneness

Throughout this section, the dependent variable is change proneness (see Section 2.3.1). The independent variables include the size and static coupling measures (defined in Appendix A.1 and A.2, respectively), and our proposed 12 dynamic coupling measures. Ordinary Least-Squares regression (including outlier analysis) is used to analyse and model the relationship between the independent and dependent variables, that is, between the size/coupling measures of the first sub-release and the amount of changes in the subsequent sub-releases. In order to select covariates in our regression model, we use a mixed selection heuristic [FW98] so as to allow variables to enter, but also to leave, the model when below/above a significance threshold. Though other procedures have been tried (e.g., backward procedure based on variables with highest loadings in principal components), the one we report here yielded models with significantly higher fit.

Recall that the objective of this regression analysis is to determine whether dynamic coupling measures help to explain additional variation in change proneness, compared to class size and static coupling alone (see Section 2.3.1). In other words, we want to determine whether these measures help to obtain a better model fit and, therefore, an improved predictive model. To achieve this objective we proceeded in two steps. First we analysed the relationship between *Change* and *CS + Static* coupling measures in order to generate a multivariate regression model that would serve as a baseline of comparison. We then continued by performing multivariate regression, using as candidate covariates all size, static coupling, and dynamic coupling measures. If the goodness of fit of the latter model were significantly better than the former model we would be able to conclude that dynamic coupling measures are useful, additional explanatory variables of change proneness.

The first multivariate model we obtained when using size and static coupling measures as candidate covariates is presented in Table 2.4 4. After removing one outlier that is clearly over-influential on the regression results (with an extremely large Change value), we obtained a model with three size measures and nine static coupling measures for covariates (for 135 observations). Around 79% of the variance in the data set is explained by size and static coupling measures and we obtained an adjusted R^2 of 0.77 (i.e., adjusted for the number of covariates [FW98]). We do not attempt to discuss the regression coefficients, because such models are inherently difficult to

Table 2.4: Regression model using size and static coupling measures as candidate covariates

Covariate	Coefficient	Std Error	t Ratio	$Prob > \|t\|$
Intercept	14.246195	4.085184	3.49	0.0007
CBO	2.8096468	0.787344	3.57	0.0005
PIM_EC	1.4540822	0.22098	6.58	$< .0001$
DAC'	18.873312	4.379724	4.31	$< .0001$
OCAEC	-5.362183	2.456952	-2.18	0.0310
ACMIC	-26.6476	6.442303	-4.14	$< .0001$
OCMIC	-12.6526	1.017126	-12.44	$< .0001$
OMMIC	4.2143694	0.507107	8.31	$< .0001$
DMMEC	-2.996934	0.583487	-5.14	$< .0001$
OMMEC	-1.4191	0.344409	-4.12	$< .0001$
NMD	-1.066682	0.288966	-3.69	0.0003
NumPara	4.2301027	0.389313	10.87	$< .0001$
CS2 (semi)	-0.373677	0.037926	-9.85	$< .0001$

interpret since it is common to see some degree of correlation (as shown by the PCA) and interaction between covariates. Smaller, less accurate models (e.g., where covariates are selected based on principal components) would have easier to interpret but recall that our goal was to demonstrate the usefulness of dynamic coupling measures as predictors of change proneness. Furthermore, the analysis in Section 2.3.6 has shown that, when significant, the relationships are in the expected direction for our dynamic coupling measures.

When including, in the set of candidate covariates, the dynamic coupling measures, we obtain a very different model (Table 2.5). Four dynamic coupling measures, as well as nine static coupling measures and four size measures, were included as covariates in the model (we retained, as for the other model, all covariates with p-values below 0.1). The model explains 87% of the variance in the data set and shows an adjusted R^2 of 0.85. Therefore, even when accounting for the difference in number of covariates, the coefficient of determination (R^2) increased by 8% or 35% of the unexplained variance (from 0.77 to 0.85) when using dynamic coupling measures as candidate covariates. This is an indication that some of the dynamic coupling measures are complementary indicators to static coupling and size measures as far as change proneness is concerned.

It is also interesting to note that three out of the four dynamic coupling measures capture export coupling. One import coupling measure is nevertheless selected, but is clearly less significant. One explanation is that, from the PCA in Section 2.3.5[3], we can see that *class-level* dynamic coupling measure

[3]Simple correlation analysis indicated the same phenomena, though for the sake of brevity, this is not reported here.

Table 2.5: Multivariate regression model using all measures as candidate covariates

Covariate	Coefficient	Std Error	t Ratio	$Prob > \|t\|$
Intercept	8.1297083	3.629	2.24	0.0270
EC_OC	4.3269164	1.082985	4.00	0.0001
EC_OM	-7.701119	1.59995	-4.81	$< .0001$
EC_OD	5.0275368	0.999278	5.03	$< .0001$
IC_CC	-1.144599	0.522802	-2.19	0.0306
CBO	2.842625	0.707569	4.02	0.0001
RFC_1	0.6700376	0.183078	3.66	0.0004
RFC	-0.058428	0.017346	-3.37	0.0010
OCAIC	19.390073	4.236142	4.58	$< .0001$
OCMIC	-10.37291	0.951485	-10.90	$< .0001$
OMMIC	4.373865	0.553446	7.90	$< .0001$
DMMEC	-1.170562	0.425857	-2.75	0.0069
OMMEC	-1.462298	0.25477	-5.74	$< .0001$
AMAIC	6.0653471	1.98776	3.05	0.0028
NMI	4.386547	0.982219	4.47	$< .0001$
NMpub	-1.868996	0.589443	-3.17	0.0019
NumPara	2.6044268	0.738117	3.53	0.0006
CS1 (SLOC)	-0.226613	0.023067	-9.82	$< .0001$

tend to be more correlated to size and static coupling and, similarly, dynamic *export* coupling measures tend to be less correlated to size measures than their *import* counterpart. A likely reason is that it is easy to imagine small classes providing services to many other methods and therefore having a large export coupling. Large import coupling classes though, are more likely to be large, because they use many features from other classes. Results in our earlier study on a Smalltalk system [Ari02] also showed that dynamic export coupling is a stronger indicator of change proneness. Though the context, programming language, and application domain were different, it is interesting to note that the result obtained in the two studies are consistent, thus suggesting our results can be generalised to a large proportion of systems.

2.4 Related Works

Dynamic object-oriented coupling measures were first proposed in [YAR99]. The authors proposed two object-level dynamic coupling measures, Export Object Coupling (EOC) and Import Object Coupling (IOC), based on executable Real-Time Object-Oriented Modelling (ROOM) design models. The design model used to collect the coupling measures is a special kind of sequence diagram that allows execution simulation.

IOC and EOC count the number of messages sent between two distinct

objects o_i and o_j in a given ROOM sequence diagram x, divided by the *total* number of messages in x. Thus, the result is a percentage that reflects the "intensity" of the interaction of two objects related to the total amount of object interaction in x. For example, in a simple scenario $x1$ where o_1 sends two messages ($m1$ and $m2$) to o_2 and o_2 sends one message ($m3$) to o_1, then $IOC_{x1}(o_1, o_2) = 100 * 2/3 = 66\%$ and $IOC_{x1}(o_2, o_1) = 100 * 1/3 = 33\%$. Based on these basic measures, the authors also derive measures at the system level using the probability of executing each sequence diagram as a weighting factor. In a different paper, a methodology for architecture-level risk assessment based on the dynamic measures is proposed [YAR00].

There are several important differences between the measures presented in [YAR99] and the coupling measures described in this paper:

- The dynamic coupling measures in [YAR99] do not adhere to the coupling properties described in [BDW99]. This is not necessarily a problem in the application context of that particular piece of work, but it would very likely be a problem in many other situations (see [BDW99] for a detailed discussion).

- The measures described in this paper differentiate between many different dimensions of coupling, in addition to import and export coupling. Most importantly, we account for inheritance and polymorphism by distinguishing between dynamic class-level and object-level measures. In our opinion, the ability to measure coupling precisely for systems with inheritance and dynamic binding represents one of the primary advantages of dynamic coupling over static coupling. This is supported by the results presented in the previous section.

- Our measures are collected from analysing message traces from system executions (Section 2.3.2) or from UML diagrams (Section 2.2.4). The dynamic coupling measures in [YAR99] are collected from ROOM models.

Another important addition to [YAR99] is that we perform an empirical validation of our dynamic coupling by showing they are complementary to simple size measures and static coupling measures. Furthermore, their relationship to an external quality indicator (change proneness) is investigated.

The measures proposed and validated in this paper are based on an initial study described in [Ari02]. Initially the dynamic coupling measures were described informally, and an initial validation was performed on a SmallTalk system. In the current paper, this research has been extended in several important ways. The dynamic coupling measures have been defined formally and precisely, in an operational form. As part of this process, we discovered that some of the measures proposed in [Ari02] did not fully adhere to the coupling properties described in [BDW99]. The measures proposed in this

paper are shown to be *theoretically* valid, at least based on a widely referenced axiomatic framework. The *empirical* validation in this paper is also considerably more comprehensive than in [Ari02]. In the current paper, the dynamic coupling measures are compared with size and static coupling measures. Such a comparison was not possible for the SmallTalk system investigated in [Ari02] because static measures could not be collected. This paper clearly confirms the initial empirical evaluation described in [Ari02]; both in terms of Principal Component Analysis and evaluation of the dynamic coupling measures as predictors of change proneness. Thus, the two studies provide a strong body of evidence that the proposed dynamic coupling measures (especially export coupling) are useful indicators of change proneness and capture different properties than do static coupling measures. Results were found to be very similar (despite some differences in measurement) across two separate application domains (commercial CASE tool and open-source web software, respectively) and programming languages (SmallTalk and Java, respectively).

2.5 Conclusion

The contribution of this paper can be summarised as follows. Firstly, we provide formal, operational definitions of dynamic coupling measures for object-oriented systems. The motivation for those measures is to complement existing measures that are based on static analysis by actually measuring coupling at run-time in the hope of obtaining better decision and prediction models, because we account precisely for inheritance, polymorphism and dynamic binding. Secondly, we describe a tool whose objective is to show how to collect such measures for Java systems effectively and, finally yet importantly, we perform a thorough empirical investigation using open source software. The objective was three-fold: (1) Demonstrate that dynamic coupling measures are not redundant with static coupling measures, (2) Show that dynamic coupling measures capture different properties than simply size effects, (3) Investigate whether dynamic coupling measures are useful predictors of change proneness. Admittedly, many other applications of dynamic coupling measures can be envisaged. However, investigating change proneness was used here to gather initial but tangible evidence of the practical interest of such measures.

Our results show that dynamic coupling measures indeed capture different properties than static coupling measures, though some degree of correlation is visible, as expected. Export coupling measures were shown clearly to be strongly related to change proneness, in addition to that which can be explained by size effects alone. Lastly, some of the dynamic coupling measures, especially the export coupling ones, appear to be useful, complementary indicators of change proneness when combined with size and static coupling

measures. Some of these results confirm those obtained on an earlier study [Ari02] of a SmallTalk system. Though no comparison with static coupling and size measures could be performed in this earlier study, those combined results constitute strong evidence that dynamic export coupling measures are strong indicators of change proneness.

Future work will include investigating other applications of dynamic coupling measures (e.g., test case prioritisation), and the cost-benefit analysis of using change proneness models such as the ones presented in the current work. These models may be used for various purposes, such as focusing supporting documentation on those parts that are more likely to undergo change, or make use of design patterns to better anticipate change.

Acknowledgements

Many thanks to Magne Jørgensen, Vigdis By Kampenes, Amela Karahasanovic, Dag Sjøberg, Kristin Skoglund, Ray Welland and Jürgen Wüst for valuable contributions to the research presented in this paper. Lionel Briand was partly funded by an NSERC operational grant and a Canada Research Chair.

References

[ABF03] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. Technical report, Simula Research Laboratory, TR 2003-5/Carleton University, Canada, TR SCE-03-18, 2003.

[ABFar] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, To Appear.

[Ari01] E. Arisholm. *Empirical Assessment of Changability in Object-Oriented Software*. PhD thesis, University of Oslo, Oslo, 2001.

[Ari02] E. Arisholm. Dynamic coupling measures for object-oriented software. In *proc. 8th IEEE Symposium on Software Metrics (METRICS'02)*, pages 33–42. IEEE Computer Society, 4-7 June 2002.

[AS00] E. Arisholm and D. I. K. Sjøberg. Towards a framework for empirical assessment of changeability decay. *The Journal of Systems and Software*, 53(1):3–14, 2000.

[ASF04] The Apache Software Foundation. The Apache Jakarta project. http://jakarta.apache.org/, 2004.

[ASJ01] E. Arisholm, D. I. K. Sjøberg, and M. Jørgensen. Assessing the changeability of two object-oriented design alternatives - a controlled experiment. *Empirical Software Engineering*, 6:231–277, 2001.

[BAJ01] L. Bratthall, E. Arisholm, and M. Jørgensen. Program understanding behaviour during estimation of enhancement effort on small Java programs. In *proc. PROFES 2001 (3rd International Conference on Product Focused Software Process Improvement)*, 2001.

[BDM97] L. C. Briand, P. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In *proc. 19th International*

Conference on Software Engineering (ICSE'97), pages 412–421, 1997.

[BDW98] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[BDW99] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan./Feb. 1999.

[BeA95] F. Brito e Abreu. The MOOD metrics set. In *proc. ECOOP'95 Workshop on Metrics*, 1995.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Users Guide*. Addison-Wesley, 1998.

[BS98] A. B. Binkley and S. R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *proc. 20th International Conference on Software Engineering (ICSE'98)*, pages 452–455, 1998.

[BW02a] L. C. Briand and J. K. Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 59:97–166, 2002.

[BW02b] L. C. Briand and J. K. Wüst. The impact of design properties on developement cost in object-oriented systems. Technical Report TR-99-16, ISERN, 2002.

[BWIL99] L. C. Briand, J. K. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality in object-oriented designs: an industrial case study. In *proc. 21st International Conference of Software Engineering (ICSE'99)*, pages 345–354, 1999.

[BWL99] L. C. Briand, J. K. Wüst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *proc. International Conference on Software Maintenance (ICSM'99)*, pages 475–482, 1999.

[CDK98] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–637, 1998.

[CG93] R. E. Courtney and D. A. Gustafson. Shotgun correlations in software measure. *Software Engineering Journal*, pages 5–13, Jan. 1993.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CKK+00] M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. In *proc. Fourth Euromicro Working Conference on Software Maintenance and Reengineering*, pages 45–54, 2000.

[CS00] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Systems*, 26(8):786–796, 2000.

[DBM+96] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[DSWR02] I. S. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis. A review of experimental investigations into object-oriented technology. *Empirical Software Engineering*, 7(3):193–232, 2002.

[Dun98] G. Dunteman. *Principal Component Analysis*. SAGE publications, 1998.

[EEBGR01] 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.

[FW98] R. J. Freund and W. J. Wilson. *Regression Analysis: statistical modelling of a response variable*. Academic Press, 1998.

[HCN98] R. Harrison, S. J. Counsell, and R. V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[HHL90] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the maintainability of object-oriented software. In *proc. IEEE Region 10 Conference on Computer and Communication Systems (TENCON'90)*, pages 404–409, 1990.

[LH93] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 1993.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.

[YAR99] S. M. Yacoub, H. H. Ammar, and T. Robinson. Dynamic metrics for object-oriented designs. pages 60–61, 1999.

[YAR00] S. M. Yacoub, H. H. Ammar, and T. Robinson. A methodology for architectural-level risk assessment using dynamic metrics. In *proc. 11th International Symposium on Software Reliability Engineering*, pages 210–221, 2000.

Chapter 3

Jdissect - a Dynamic Coupling Tracer for Object-Oriented Systems

This chapter describes Jdissect, the tool created to obtain the coupling measures presented in Chapter 2. The first section gives an overview of the coupling tracer and its relation to Java. Having treated the basic outline of Jdissect, we move on to provide a more detailed account of its design. Following the section on design is a short tutorial on practical use, and a description of the Velocity case study. The chapter ends with a discussion of various technical choices and trade-offs made during the implementation of Jdissect.

3.1 Overview

One of the goals of this research project is to investigate how coupling impacts change-proneness in software. The previous chapter describes twelve measures which can be used to quantify coupling in most object-oriented programming languages. The approach used differs from traditional methods in that we want to explore dynamic coupling, i.e., coupling data from a running program.

The tool which collects dynamic execution information and determines coupling is called Jdissect. Jdissect interfaces with a running Java program and registers its execution history. This data can later be employed to calculate the proposed dynamic coupling measures.

3.1.1 Java

Before we go on to portray the workings of Jdissect in detail, it is necessary to explain some of the fundamental properties of Java which enable gathering

Table 3.1: Java interfaces to native code

Acronym	Name	Use
JNI	Java Native Interface	Interfacing with native libraries.
JVMPI	Java Virtual Machine Profiling Interface	Creating Java profilers using native code.
JVMDI	Java Virtual Machine Debugging Interface	Creating Java debuggers using native code.

dynamic execution information.

Computer programs are usually created by writing source code in some programming language. There are many popular languages, such as C, C++ and Java. Source code in these languages needs to be compiled before it can be executed. Compilation involves translating the relatively abstract source code representation of a program into machine code understood by a computer's CPU (Central Processing Unit). This is often referred to as "native code", as it is directly comprehensible to the native CPU of the computer. Interfacing with, and collecting information from, an executing native code program is difficult. It requires a deep understanding of what goes on inside the CPU itself. Furthermore, different CPU types use widely varying machine code, and have very different capabilities.

Java is not like most ordinary programming languages. The initial steps to create an executable application are similar to those of C and C++. However, the similarity ends when the program is compiled. Instead of translating source code into native machine code during compilation, the Java compiler transforms source code into byte code. In short, byte code is very similar to machine code, except for the fact that it is not tied to any specific CPU type or model. Byte code represents an artificial instruction set, much like that used in the `MIX` and `MMIX` machine languages presented by Knuth [Knu97]. To execute programs in byte code format, a separate program is required. In the case of Java, this program is called the Java Virtual Machine (JVM). The JVM emulates a processor supporting the byte code instruction set.

One advantage of using the JVM is that it supports interfacing with external libraries during execution of Java programs. The intention behind this capability is to allow inspection of a running program. This makes it possible to create efficient debugging and profiling tools. In addition, the JVM has an interface which enables developers to call libraries containing native code from within Java programs.

Jdissect needs to obtain information on what happens inside a Java program as it executes. For large and long running programs the amount of information collected is potentially huge. In order not to impact perform-

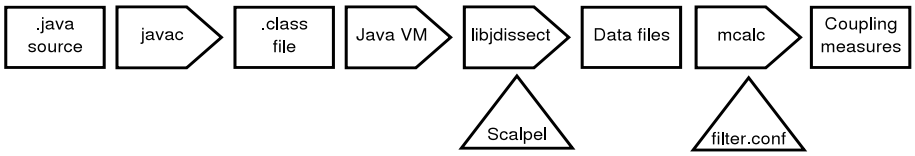

Figure 3.1: Data flow – from Java source file to coupling measures

ance too severely (see Section 3.6.3), and because of the relatively easy access to information, Jdissect employs the native code interfaces provided by the JVM. Table 3.1 lists the interfaces used, and the tasks they were originally intended for.

3.1.2 Jdissect

Jdissect consists of a library and an executable program, both written in C++. The Jdissect library, `libjdissect.so`, collects data from the Java Virtual Machine by using the three interfaces shown in Table 3.1. The process of calculating coupling from the data stored by `libjdissect.so` is performed by a program named `mcalc`.

Figure 3.1 depicts the flow of information from source code to calculated coupling measures. Arrows symbolise information flow, while boxes represent stored information. The triangles denote user interaction and configuration.

The first phase of the process involves compiling the source code (`.java` files) of the applications which is to be examined into executable byte code form (`.class` files). Next, the application is executed by the Java VM. The JVM sends information to the Jdissect library with details of significant events. For instance, method calls and class instantiation. During this phase the user has the opportunity to "tag" sections of the execution history, using a separate program called `Scalpel`, described in Section 3.3.1. Tagging can be employed to mark messages as occurring in specific use-cases or during use of certain functional units. Once the Java application terminates the collected information is stored on disk.

The coupling measures are calculated in the second phase of the analysis process. First, `mcalc` loads the data stored during one or more Java VM sessions. The loaded data is then filtered according to the rules found in the `filter.conf` configuration file. Finally, the coupling measures described in Chapter 2 are calculated and displayed.

3.1.3 Data - Aggregation and Filtering

Data starts out as collections of files representing the execution history of an application in one or more instances of the Java VM (marked "data files" in Figure 3.1). The Jdissect data collection library, `libjdissect.so`, does not

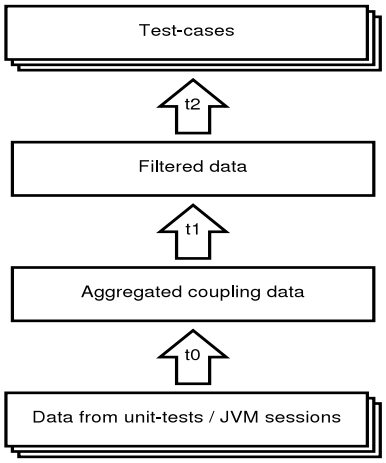

Figure 3.2: Data aggregation and filtering

remove any of this information as it is stored on disk. Determining exactly what data should be retained when the dynamic coupling measures are calculated is therefore up to `mcalc`. As a result, `mcalc` is more complicated and requires more in the way of configuration than `libjdissect.so`.

The steps in the process of reading and filtering data can be seen as a series of conceptual layers which information passes through before the coupling measures are calculated. There are four such layers, illustrated in Figure 3.2. While the layers themselves hold data at various stages, the transition from one layer to another, $(t_0 \cdots t_2)$, is where active steps can be taken to manipulate and filter the data. This enables users of `mcalc` to focus on calculating coupling for specific components, libraries, frameworks or functional units of the application being analysed.

What follows is a description of the layers, and what occurs during the transitions between them.

Aggregation - t_0

In most usage scenarios, an entire Java application can not be tested in a single session with the JVM. This is because exercising an application to obtain satisfactory degrees of code coverage, as described in Section 3.5.2, frequently requires executing multiple unit-tests. Each such test is usually executed in a separate instance of the Java Virtual Machine, and will leave its own dataset stored on disk.

The first transition, t_0, occurs as `mcalc` loads one or more datasets. If more than one dataset is loaded, the aggregated data will represent the union of *all* the specified execution histories. This allows aggregation of data from separate executions, or even completely unrelated applications.

Filtering - t_1

The next transition, t_1, makes it possible to focus on specific areas of an aggregated dataset. Applications often make extensive use of external libraries and frameworks. In many instances this information will distort the perspective sought by the user.

For example, starting any Java application, no matter how small, will result in massive amounts of data concerning coupling to, and between, system library classes. This might make it difficult to discern coupling between important core classes of an application from coupling to the system libraries. By providing `mcalc` with a filter configuration, the user can determine which libraries and classes should be accounted for when coupling is calculated.

Grouping Based on Functional Unit - t_2

An application is composed of functional units; entities capable of accomplishing a specified purpose. Operations like "save customer to file" and "load customer from file" are typical examples of functional units.

The exact level of detail used in the definition of functional units is decided by the user. They can for example represent a single UML use-case, or an arbitrary set of operations perceived to be related. In some instances it is desirable to group data according to functional units, and to examine coupling data for one such unit at a time. Java, however, has no concept of what a functional unit consists of, and does not recognise, for example, UML use-cases. Users of `libjdissect.so` will therefore have to add this information by "tagging" sections of execution history as a *test-case*. This process is described in Sections 3.1.2 and 3.3.1.

A user might, for example, tag the "load customer from file" operation and examine coupling within the confines of that single functional unit. The tagging operation results in meta-information being added to the data collected by `libjdissect.so`. This test-case meta-information can later be employed to locate specific functional units within an application's execution history.

However, when calculating coupling, the transitions t_0 and t_1 do not enable `mcalc` to group data according to test-cases. The last transition, t_2, solves this problem. When data has been aggregated and filtered (t_0 and t_1), the remaining data undergoes one last transition; t_2. During this process the user can choose to remove any parts of an application's execution history which has not been tagged with the appropriate test-case meta-information. For example, if a user has tagged the functional unit "load customer from file" during data collection, `mcalc` can be instructed to only calculate coupling for this part of the execution history.

3.2 Design

This section explains some of the deliberations behind the design of Jdissect. First, we focus on *why* and *how* the same core data structure is reused in both `libjdissect.so` and `mcalc`. Next, some arguments in favour of Jdissect's data model are presented. The section ends with an explanation of two classes which are important in assembling and retaining the data structure.

3.2.1 Overview

On the surface, `libjdissect.so` and `mcalc` may seem to perform wholly unrelated tasks. The library is responsible for collecting and storing data, while `mcalc` calculates coupling. However, if the various functions performed by the two programs are broken down into smaller pieces and compared, a number of similarities appear.

Both components need the following functionality:

1. Initialise data structure based on some source.

2. Access the structure in a consistent manner.

3. Use the data in the structure to perform some action.

On account of these similarities, a substantial amount of code is shared between the two seemingly separate tools. Most important is the fact that they use the same structure to retain data. Consequently, both programs reuse not only the implementation of the structure itself, but also the mechanisms used to access and update it.

There are important benefits to be had from reusing code in this manner. Creating one common interface instead of two separate ones saves time with respect to implementation. In addition, if errors are discovered in the shared functionality, corrections will only have to be applied once.

Figure 3.3 shows the relationship between the two programs using UML notation. *ModelBuilder* and *SetContainer* are classes in the "Set utilities" package. The library and analysis application are depicted as packages [AN02]. The package called "core data-model" represents the class diagram presented both in Chapter 2 and in Figure 3.4.

The two classes are shared between the packages containing the specialised functionality of the two programs. Both the data collection library and `mcalc` assemble the data model by utilising the functionality in *ModelBuilder*. They subsequently use the methods in *SetContainer* to access the information in a manner which is consistent with the definitions of sets in Chapter 2.

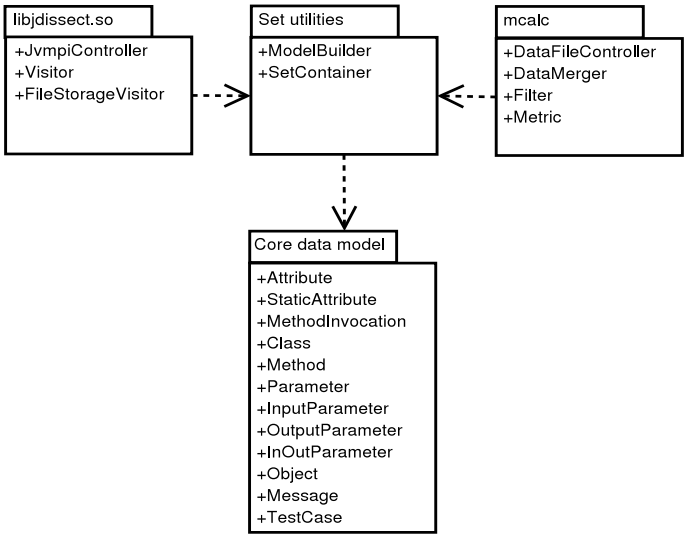

Figure 3.3: Classes and subsystems

3.2.2 Core Model

In Chapter 2, we explained how an object-oriented program and its execution history can be represented by set-theory elements. For example, a message between two objects is uniquely defined by the binary relation $ME \subseteq O \times M \times N \times O \times M$, where ME is the set of all such relations. The implementation of the coupling measures is based on analysing these sets and the relations between their elements.

Figure 3.4 presents the class diagram of the core data model employed by Jdissect in more detail. It is worth noting that some class names differ slightly from the names used in the previous chapter.

The model is not specific to Java. It can be used to represent the execution of almost any object-oriented program. However, doing so would require small changes to some of the multiplicities in the UML class diagram and to the source code. For example, to support a language where multiple inheritance is possible (e.g., C++), the multiplicity of the ancestor relationship used in the definition of *Class* would need to be changed from 1 to $0..*$.

Justification

The data structure presented in Figure 3.4 may seem overly complex in comparison to the task it is used to accomplish. For example, none of the coupling measures presented in Chapter 2 use *Parameter* or *Attribute* in their definition. The same argument can be made against many of the relationships between classes in the model; they are not actively employed to

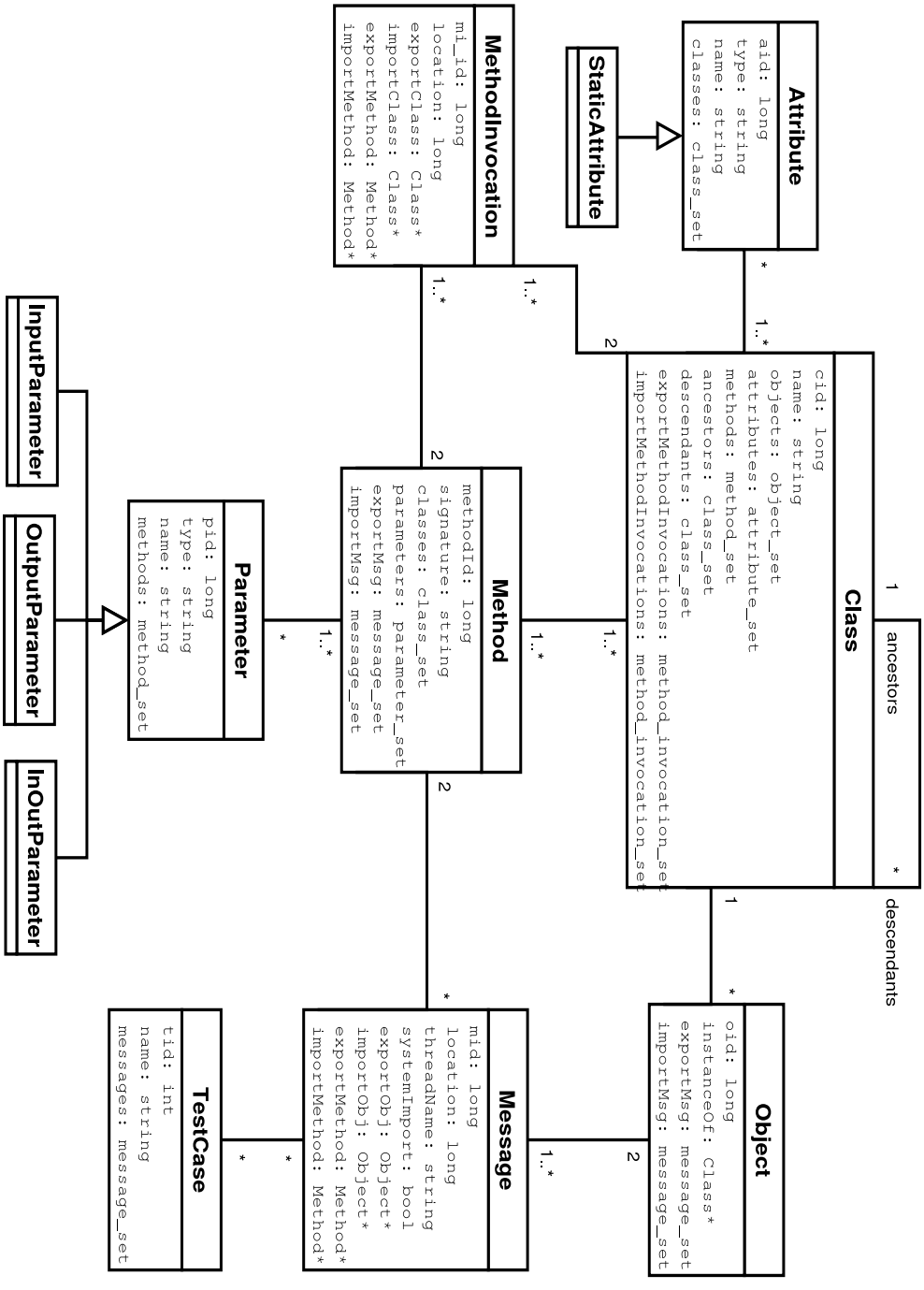

Figure 3.4: Core data model used in Jdissect.

determine coupling, and are therefore redundant. A less complicated structure would mean decreased memory consumption, and possibly an increase in execution speed.

There are, however, several good reasons for retaining this model. One of the primary arguments is that the model is already very close to the theoretical definitions used in the coupling measures. This has the advantage of making implementation of the defined measures trivial once the theory is understood, hence increasing the likelihood of a correct implementation. It also makes incorporating changes and corrections in the theoretical definitions much easier, as the relationship between the conceptual and actual model is one-to-one.

Another argument in favour of this model, in its present generic form, is that creating new measures is easy. The model implementation stores nearly all the data it receives from the Java VM. Had the design been optimised solely for the measurement definitions presented in Chapter 2, new measures might have required re-implementing large sections of the model and the surrounding logic. As the model is currently defined, implementing new measures only requires slightly different analysis methods.

The flexibility of the core model turned out to be an advantage on two occasions during work on this case study and the resulting paper.

1. During the implementation of Jdissect work on the measurement definitions was still ongoing. At one point we found that our original definition of a message ($ME \subseteq O \times M \times O \times M$) was not adequate. We had not recognised the importance of the line number from which a method call originates. The correct binary relation is $ME \subseteq O \times M \times N \times O \times M$ instead. Making the necessary changes was more a problem of obtaining the needed data from the underlying Java APIs, than of changing the model itself.

2. After the completion of the case study and article, the IEEE review committee suggested a number of improvements. One of these involved documenting the number of overridden methods and inheritance relationships in Velocity. Performing this count manually would have been both time consuming and prone to errors. Instead, a new type of measure was implemented. Creation and verification of the new measure within the existing framework took only one hour. This shows that the core data structure, in combination with the surrounding framework, is both versatile and flexible.

The *TestCase* Class

It is sometimes desirable to examine data pertaining to specific operations within a given time interval. This is typically the case if we want to examine

coupling in specific UML use-cases, independent subsystems, and in general; anytime we wish to analyse specific functional units by themselves.

One way of accomplishing this is by restarting the data collection procedure for each use-case or feature. However, this does not work well in situations where multiple use-cases or features depend on each other. Additionally, restarting an entire Java application to gather data from individual features is a tedious task.

TestCase represents meta-information which can be used to group messages according to criteria specified by the user. The class is dissimilar to the other classes in the core model in that it does not directly represent any part of a running program. It is rather an attempt at adding a notion of functional units to the execution history of a program.

All instances of the *Message* class are related to one or more *TestCase*s. This enables us to, for example, add a new test-case each time we start executing a new feature or use-case. It is subsequently possible to obtain independent sets of coupling measures from each specific test-case.

3.2.3 The *ModelBuilder* Class

The core model plays a central role in both data collection and subsequent analysis. In the course of obtaining and analysing coupling data it is assembled no less than three times from various sources. The following list shows where the model is populated.

1. `libjdissect.so` – Created based on data received from the Java Virtual Machine during a session.

2. `mcalc` – Reconstructed from files stored by `libjdissect.so`.

3. `mcalc` – Assembled from data stored during multiple Java sessions.

In `mcalc` the model is used twice. First as data is read from disk, and then a second time to merge the newly loaded model with one representing data from multiple JVM sessions (see Section 3.1.3).

Unfortunately, populating the model with data is not trivial. This is caused by the fact that relationships between classes are bi-directional, and that there are many inter-class dependencies which must be upheld when new data is added to the model. For example, adding a new *Attribute* requires the model to already hold the *Class* containing it. The *Class* must add a reference to the *Attribute* in its `attribute_set`, and vice versa.

Had Jdissect only initialised its core data model in one place, the complexity of these relationships might not have been a cause for a more elaborate design. However, as the core model is assembled from three different sources, it is important to encapsulate and reuse the procedure.

The class made responsible for assembling the core model is called *ModelBuilder*. It is reminiscent of the "builder" pattern presented in [GHJV94]. The stated purpose of this pattern is to *"separate the construction of a complex object from its representation so that the same construction process can create different representations"* [GHJV94, p. 97].

The builder pattern is often used to create different representations of an underlying structure. For example, to read a text document in one format and output a wide range of others. Jdissect needs the opposite of this, namely to build the exact same structure from different sources.

ModelBuilder deviates from the pattern definition in that it does not create "different representations" of the data. The class does, however, decouple assembly of the core model from the parts used in its construction.

3.2.4 The *SetContainer* Class

Chapter 2 introduced five sets which were subsequently used as building blocks in defining the coupling measures. These sets were C (classes), O (objects), M (methods) , ME (messages) and IV (method invocations). In addition, "lines of code" was defined on the set of natural numbers, $\mathbb{N}$.

While these sets are the only ones needed to describe the coupling measures, six additional sets are defined in correspondence with the other classes in the core model.

- P - The set of function parameters (as identified by their name and type). P can be partitioned into the three subsets I, OU, IO, with $P = I \cup OU \cup IO$ and $I \cap OU \cap IO = \varnothing$.
  - I, the set of input parameters.
  - OU, the set of output parameters.
  - IO, the set of input/output parameters (e.g., references in C++).
- A - The set of class attributes in the system (identified by their name).
- SA - The set of static class attributes in the system (identified by their name).

There are several requirements relating to accessing the sets and elements of the core model. First, implementation of the coupling measures relies on convenient access to the different sets. Next, storing the core model to disk requires functionality for traversing the structure in order. In addition, calculating some of the coupling measures requires access to derived sets like IV.

Adding this functionality to *ModelBuilder* would have given it more than one role. It would have become responsible for building the structure, traversing it and accessing both derived and ordinary sets. Giving a class

many different roles would have gone against the principle of only assigning strongly related responsibilities and functions to a class (cohesion).

Instead of delegating these functions to *ModelBuilder* we created *SetContainer*. It is responsible for access to all the sets and for providing some of the functionality needed to traverse the core model. Derived sets, like IV, are accessed in the same manner as any other sets. The process of deriving them is completely encapsulated in *SetContainer*.

3.3 Jdissect - Collecting and Analysing Data

This section explains how Jdissect is used to obtain and analyse data. It starts with an overview of the different options needed to ensure that the library is loaded by the Java Virtual Machine. Next follows a description of the data analysis program `mcalc`. The usage-tutorial ends with a discussion of the format and rules employed in the configuration file used by `mcalc`.

3.3.1 Collecting Data - `libjdissect`

In order to collect data from a Java application the JVM must load the Jdissect library on starting. Otherwise, the Java program will execute normally, and no data will be collected. Given that an application named `MyJavaApplication` is compiled in the directory `/tmp/jdissect`, this can be accomplished with the following commands in a Unix shell like `bash` or `sh`. This command invokes a version of the Java VM located in the directory specified by `$JAVA_HOME`. The four different options are explained in turn.

Listing 3.1: Starting a Java application with Jdissect

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/jdissect

$JAVA_HOME/bin/java -Xrunjdissect:/tmp/jdissect/data \
                    -Xdebug -Xnoagent -Xjava.compiler=NONE \
                    /tmp/jdissect/MyJavaApplication
```

The backslash (\) means that the command continues on the next line. This character is removed by the shell prior to execution.

The `-Xrun` Option

The `-Xrun` option is used to load debugging or profiling libraries meant to interface with the VM. While Jdissect does not belong in either category, it depends on using both profiling and debugging functionality in the Java APIs.

The exact format of the `-Xrun` option is:

Listing 3.2: The `-Xrun` option

```
-Xrun<library name>[:<options>]
```

The first part of the argument, indicated by the characters '<' and '>', is required. Everything enclosed in '[' and ']' is optional, including the colon. However, if the colon is included, arguments are expected. The library name argument is automatically given the prefix "lib" and the suffix ".so". So, even though the argument in the above example is only `jdissect`, it will cause Java to attempt locating a library called `libjdissect.so`.

If loading Jdissect is to be successful the operating system's dynamic linking loader must be able to locate the library. On Linux systems the loader starts by searching paths indicated by the environment variable `LD_LIBRARY_PATH`. If the library is not found in any of these directories, the loader continues to search the standard library paths, usually `/lib`, and `/usr/lib`. It is common practise to put small and highly specialised libraries such as Jdissect in their own directories, instead of cluttering the system-wide library paths. In Listing 3.1 the library is located in `/tmp/jdissect`.

Everything after the colon character is passed unmodified to the library as an argument. Jdissect requires only one argument, specifying where it should store data. In this example invocation of Java, Jdissect will store its data files in `/tmp/jdissect/data`.

Other `-X` Options

The last three options to the Java VM are related to the setup of debugging and execution. The user should feel free to experiment with different settings and options depending on the Java version and implementation vendor. These settings have been found to function best with Java implementations from IBM, versions 1.3.1 and 1.4.

One of the arguments against Java is that it interprets its own virtual machine code instructions (byte codes), rather than using the native instruction set of the architecture it runs on directly. This results in somewhat slower execution and higher overhead than applications compiled into native machine code.

In an attempt to speed up execution many Java implementations now include what is called a Just-In-Time (JIT) compiler. A JIT compiler translates Java byte codes into native instructions as a program executes, storing the native machine code for later use. The effect is that the second time a block of code executes, it is quicker. However, this creates problems for libraries such as Jdissect because some of the translated code does not trigger the usual debugging and profiling events. This leads to loss of information. In some instances it even causes Jdissect to crash. It is therefore not a good idea to leave the JIT optimisations turned on. So in order to capture the best possible data we add the options `-Xdebug -Xnoagent -Xjava.compiler=NONE`. The `-Xjava.compiler` option turns off JIT compilation, while the other two options configure debugging.

Collecting Data in Practise

Listing 3.1 shows how to collect data from a single program. It is not a particularly realistic example as most applications have hundreds of features and options which often require some form of user input or interaction.

If the collected data is to be an accurate representation of an application, it is important that as much as possible of the application's code is executed. We refer to this as "code coverage". Typically we aim for levels of coverage above 90% when collecting data in practise. The most convenient way to accomplish this is by using unit- or regression-tests written during development of the application.

Such tests are often only small fragments of code, each designed to exercise a small subset of functionality or a specific feature. When analysing an application we want the opportunity to examine the whole, not just separate parts.

One method of using unit-tests to collect data is by executing each of them in turn, while storing the obtained data in separate directories. At a later stage in the analysis process this information can be merged by `mcalc`.

Larger Java projects often employ such frameworks as JUnit in combination with Ant [ASF04a]. JUnit [GB04] is a framework for creating automatic unit-tests, while Ant is a build tool written in Java. If tools such as these are used, and the unit-tests are available, it is often possible to modify the build-file used to execute the tests to include Jdissect. An example of an XSLT stylesheet used to modify the execution environment of the Velocity unit-tests can be found in Appendix C.4.

Scalpel - Adding Test-cases Interactively

As the Jdissect data collection library starts, it attempts to open a network socket on port 9898 of the computer it is running on (localhost). If a server socket is found, the library establishes a communication channel with the host program.

This socket can be used to interactively add test-cases to the data that is gathered from an executing Java program. Test-cases can be used to group data according to application features (functional units), according to UML use-cases or based on individual unit-tests.

Included with the distribution of Jdissect is a small Java-based server called Scalpel. Scalpel has only two functions; "tag" and "quit". The "quit" function ends the program, while "tag" adds a new *TestCase* to the collected data. When the user has typed in a new tag, all subsequent messages (in practise; instances of the class *Message*) are associated with the new test-case.

This functionality was given much attention in the initial design of Jdissect. However, as deadlines approached and we found that this function

would not be used in the analysis of Velocity, it lost priority. The result is that `libjdissect.so` supports the feature, while `mcalc` at present does not.

3.3.2 Data Analysis - `mcalc`

Running a Java program with `libjdissect.so` will leave 17 different data files in the directory specified by the `[:<option>]` part of the `-Xrun` argument (see Listing 3.1). These files have little value before they are processed with `mcalc`. The `mcalc` application performs three important functions.

1. Merge data gathered from one or more unit- or regression-tests into one large in-memory data model (aggregation, t_0).
2. Remove superfluous information (filtering, t_1).
3. Group information according to test-cases, if any (grouping, t_2).
4. Calculate coupling measures based on the remaining data.

Running `mcalc`

The only input needed by `mcalc` is command line arguments specifying where it should attempt to locate data files. It also needs a configuration file called `filter.conf` in the same directory as the executable itself. A more detailed description of this file is given in Section 3.3.3.

Output from `mcalc` is divided into two different streams. One stream shows execution progress and is directed to `stderr`. The other stream is directed to `stdout` and contains the coupling measurement data.

If data from 10 different unit-tests are located in directories named `/tmp/jdissect/dataX`, where $X = 1 \cdots 10$, and the goal is to measure all these data sets, one would execute the following command:

Listing 3.3: Analysing 10 data sets with `mcalc`

```
./mcalc /tmp/jdissect/data* >measures
```

The example in Listing 3.3 redirects measurement data (`stdout`) to the file `measures`, while status messages are printed to the console. This enables the user to monitor execution progress, while leaving the measurement data in a file for further manipulation and analysis.

`mcalc` Output

Output to `stdout` from `mcalc` has a very simple structure. Each type of measure is calculated in turn for each class. The names of the different measures are included on every line in order to make further processing using a script or spreadsheet easy. Following the name of the measure is the class name, and finally the counted coupling for that measure/class combination.

Listing 3.4 shows the specification for the output format, while Listing 3.5 is an excerpt of data obtained from a small test program containing the classes *foo*, *bar* and *tst*. In Listing 3.4 `\t` refers to the tab character (ASCII 0x08), while `\n` means linefeed (ASCII 0x0A).

Listing 3.4: Format specification

```
<Measure name>\t<Class name>\t<Measured coupling>\n
```

Listing 3.5: Excerpt of output to `stdout`

```
IC_OC    tst     3
IC_OC    bar     1
IC_OC    foo     0

IC_OM    tst     5
IC_OM    bar     2
IC_OM    foo     0
```

Listing 3.6 shows output written to `stderr` as data is loaded and processed. The program displays detailed information about the size of the various sets for debugging purposes before attempting to calculate coupling.

Listing 3.6: Output written to `stderr` during `mcalc` execution

```
Merging data from path '/tmp/jdissect/'
Creating method invocation set
size(M)  1991
size(C)  303
size(O)  143
size(ME) 1618
size(IV) 624
Calculating metrics...
```

3.3.3 Configuring `mcalc`

The Jdissect library collects data under the assumption that every class, object and message is of importance. Consequently, the data files contain some information that is not needed and which might obscure interesting or important facts.

The superfluous information will often consist of classes from the Java standard libraries, classes provided by frameworks and classes used to implement unit-tests. These are typically components and messages which are not part of the core application itself. Some of this information is clearly irrelevant, and should be removed before coupling is calculated.

However, gathering data is potentially a time consuming process, and it is not always know in advance exactly what to remove. It is therefore best to postpone removal of any information for as long as possible. This is one of the reasons why filtering is done in the `mcalc` application and not in

the Jdissect library. Performing the filtering at a late stage enables easier experimentation with various settings without having to regenerate data.

filter.conf

There are two filters available in `mcalc`. The first type is called the "display filter", and controls program output. The display filter can be used to prevent coupling from being calculated for certain classes. For example, if the entire `java.lang` class-path is excluded by the display filter, no classes from this package will show up in the output from `mcalc`. However, while classes excluded by the display filter are not shown, they are still part of the data and as such they still contribute to coupling.

The second filter type is called the "count filter", and can be used to exclude both classes and whole class-paths from being counted when coupling is calculated.

Both filters are controlled from a file called `filter.conf`, located in the same directory as the `mcalc` executable.

Filter Example

As an example, consider a class A that is part of the package `test.app`. A is coupled to 6 other classes from the same package, and 10 classes from system libraries. For instance, classes in `java.io`. As we are not interested in seeing coupling data for the 10 classes which are not part of the `test.app` hierarchy, these are removed using the display filter. The result is that while the classes in `java.io` are not displayed they are still part of the coupling data for both A and the other classes in the `test.app` package. In other words; even if a class is prevented from being shown by the display filter, it is still counted when calculating coupling.

In some instances it might be useful to prevent a class from being counted. Continuing the example of A; if a user wants to measure its coupling, but does not want to count relations to any of the `java.io` classes, this can be accomplished by using the count filter.

The two filter types do not depend on each other. Thus, to prevent classes in `java.io` from being reported and counted, both filter types will need to be configured with the same exclusions.

Filter Configuration Format

The configuration files used by `mcalc` have a structure based on keywords with POSIX.2 [Ste93, pp. 26] style regular expressions as arguments. There is a total of four different keywords, two for each filter type.

The display filter is controlled using the keywords `include` and `exclude`. These keywords control the output from `mcalc`. The count filter is controlled

by the keywords `include_count` and `exclude_count`. These keywords configure `mcalc` to exclude or include various classes when measuring coupling. The configuration files can also contain comments. Lines starting with a hash-mark (`#`) are ignored, as are empty lines and lines containing only whitespace.

Each keyword can be repeated throughout the configuration file to exclude multiple system libraries and testing frameworks. The only limitation is that only one keyword can be used per line. It is common to find that the same classes should be excluded from being displayed and counted by using both filter types with the same argument strings.

As an example, consider excluding all classes from the Java system libraries with the exception of `java.util.Vector`. In other words, we do not want to see any coupling data for classes in packages starting with the prefix `java.`, with the exception of `java.util.Vector`. Neither do we want these same classes to influence calculated coupling for the rest of our application. Consequently, both the count and display filters must be employed. Listing 3.7 shows how this can be accomplished.

Listing 3.7: Practical configuration example

```
1 # Example which excludes the entire Java class hierarchy,
2 # except java.util.Vector
3 exclude java\..*
4 exclude_count java\..*
5
6 include java\.util\.Vector
7 include_count java\.util\.Vector
```

It is worth noting that a small shortcut was taken to enhance usability in the filter implementation. The regular expressions entered as arguments are prepended with the characters '`.*`'. In regexp syntax this means "match any character, zero or more times". Therefore, to exclude a class called Vector using the display filter, all the user needs to write in the configuration file is `exclude Vector`. The alternative would have been to prepend the argument with the whole class-path of Vector, or explicitly writing `.*Vector`.

It is also worth paying attention to the fact that nothing is ever appended to keyword arguments. Hence, if the desired effect is to exclude an entire package, contained in some class-path, it is necessary to explicitly specify the package name followed by '`.*`'. For example, `exclude java\..*`

Precedence Rules

Both filters are controlled using one keyword for inclusion and another for exclusion. It might be prudent to discuss the precedence rules used by `mcalc` to match class names against the configuration data.

If `mcalc` is given an empty configuration file its default behaviour is to include every class it encounters. This seems to be the most reasonable

default behaviour, as it does not presume anything about what the user wants to accomplish.

Given a configuration that contains both inclusions and exclusions, a precedence rule is needed to determine which expression are more important. The rule is simple; *inclusions have precedence*. Classes are therefore always included if they match an inclusion, even if they are excluded by another expression.

In Listing 3.7 there are exclusions which target the entire `java` hierarchy (lines 3 and 4). This means that no classes in that top-level package or any of its sub-packages are included. Further along in the configuration there are lines (6 and 7) including any class called `java.util.Vector`. The result of these two seemingly contradictory configuration instructions is that all classes in the `java` hierarchy are excluded, *except* the class called `java.util.Vector`

This precedence rule is the same for both the display and the count filter configurations. Although each type is independent of the other.

3.4 Verification of Jdissect

Before gathering data from Velocity we had to verify that Jdissect worked according to specification. Three different strategies were used to ensure conformance to the specifications and to locate possible errors and inconsistencies.

1. Store/load/store.
2. Manual verification.
3. Inter-measure symmetry properties.

These tests will be explained in turn.

3.4.1 Store/load/store Test

The first verification scheme checks if Jdissect can load and store data in a consistent manner. The rationale underlying this test is that the mechanisms involved in loading and storing data function according to specification if Jdissect can load and store the same data multiple times without introducing inconsistencies.

The first step in this test is to obtain and store data from a small test program written in Java. Subsequently, a program not dissimilar to `mcalc` is used to load the stored data, rebuild the core data model in memory, and then store it again in another directory. The result of these operations is two directories. Each supposedly containing the same data. The next step of this test is to compare the two instances of the test data contained in the

two separate directories. If there are discrepancies between them it can be concluded that Jdissect either does not build the model correctly, or that there is something wrong with the components responsible for storing data.

Performing this test we concluded that the data stored in the two directories was exactly the same. We then progressed by attempting to run the same test, but with four consecutive load/store operations. The first and last data sets were still equal. This led us to conclude that the mechanisms used to load and store data conformed to the specifications.

3.4.2 Manual Verification

The second test also starts with a Java test program. However, instead of obtaining coupling measures or data by using tools, we compute the different measures for all classes manually. Subsequently, Jdissect is used to obtain coupling measures from the test program. The two sets of measures can then be compared to see if there are any differences between them.

The test was positive as we concluded that the manual measurements were equal to those obtained by Jdissect.

3.4.3 Symmetry

Our third test is based on the observation that there are symmetry relations between coupling measures on both the class and the object level. These properties are intuitive. They basically state that any export coupling will be mirrored by import coupling at the same level (either class or object).

$$\left| \bigcup_{\forall c \in C} EC\_Cx(c) \right| = \left| \bigcup_{\forall c \in C} IC\_Cx(c) \right|, x \in \{C, M, D\} \tag{3.1}$$

$$\left| \bigcup_{\forall o \in O} EC\_Ox(o) \right| = \left| \bigcup_{\forall o \in O} IC\_Ox(o) \right|, x \in \{C, M, D\} \tag{3.2}$$

These formal expression can be used to verify inter-measure consistency. Property 3.1 can be used to determine coupling measurement integrity at the class entity level, while 3.2 has the same function at the object level.

To perform this test in practise we obtained coupling data from Velocity release 1.2-rc1, and imported it into a spreadsheet. We then calculated the sum of coupling in each direction (import/export) and at each entity level (class/object).

The results in Table 3.5 show that the sum of import and export coupling is equal for both class- and object-level measures. The success of this test leads us to conclude that the measurements obtained by Jdissect are correct.

Measure	Total coupling	Sum
IC_OC	964	3817
IC_OM	1331	
IC_OD	1522	
EC_OC	964	3817
EC_OM	1331	
EC_OD	1522	
IC_CC	728	2912
IC_CM	968	
IC_CD	1216	
EC_CC	728	2912
EC_CM	968	
EC_CD	1216	

Figure 3.5: Sum of coupling measurements for Velocity 1.2-rc1

3.5 Study of Velocity

This section contains a description of the Velocity case study. It starts by giving a short description of what Velocity is, and continues with a look at how data is collected and analysed. The section ends with an overview of the `mcalc` configuration used, and the tool used to determine code coverage.

3.5.1 Velocity

Velocity [ASF04c] is a template rewriting engine written in Java. It allows methods and variables from a surrounding Java execution environment (e.g., EJB or JSP) to be accessed from within template definitions. Data from the environment can be used in simple decision making, or merged with the template document itself. Velocity can, amongst others, be used to generate PostScript documents, web pages and SQL. It is usually employed either as a standalone utility or as an integrated component in other systems. The template specification language itself is very simple. The syntax allows only simple if-else constructs, for-each loops and variables.

Velocity is part of an ongoing effort in the Jakarta [ASF04b] community to create tools that support the MVC (Model-View-Controller) design pattern [HDFW03, p.31]. In this context, Velocity is meant to provide the functionality needed for the view.

Why Velocity?

There are several reasons why Velocity is a good candidate system for studying the relationship between dynamic coupling and change-proneness.

1. It is an open-source project under the Jakarta umbrella. This means that all source code revisions are freely available.

2. Velocity is a mature product. It has gone from version 1.0b2 released in 2001, to 1.3.1 released in 2003. All versions can be retrieved from an on-line version control system (CVS).

3. The project includes unit-tests using the JUnit [GB04] framework.

Case Study Tasks

Our case study of Velocity consists of several carefully planned phases. What follows is a summary of the different steps involved. The text in parenthesis after each task indicates the tools used to accomplish it.

1. Download the 17 different versions, from version 1-b1 to 1.3.1-rc2 from the Jakarta CVS repository. (Perl)

2. Compile the different versions and unit-tests. (Perl, Ant)

3. Perform a coverage analysis to see if the unit-tests exercised the different versions sufficiently. (Clover, Ant)

4. Determine which classes can be deemed to be "dead" code based on coverage analysis and reading the source code. (Manual)

5. Locate classes which are only used when interfacing to other systems (e.g., EJB and JSP) based on coverage, documentation and source code. (Manual, Perl, `diff, grep`)

6. Modify unit-test invocation for all Velocity versions so that they are executed with the options required by the Jdissect library, as described in section 3.3.1. Make sure data is kept in separate directories for each Velocity version. (Perl, `xsltproc`)

7. Run unit-tests with modified invocations to obtain data.
(Perl, `libjdissect.so`)

8. Create `filter.conf` configuration file based on the results of the "dead code" and interface-class analysis. (Manual)

9. Analyse data using the `mcalc` application with the configuration created in step 8. Store the results for each Velocity version. (Manual, `mcalc`)

10. Rewrite `mcalc` output to row/column order, which is more convenient for statistical analysis. (Perl)

The majority of this work is straightforward and very simple. Only item 3 and 8 require some additional mention. They will be explained in turn.

3.5.2 Measuring Code Coverage

When a Java application is executed with Jdissect all events related to loading classes, instantiating objects and calling methods are passed on to, and registered by, the Jdissect library. In itself this is not enough to create an accurate picture of what goes on inside the program.

The problem is that any application contains a multitude of logical branches. Some of them are in use quite often, while others, perhaps dealing with uncommon situations or errors, are seldom executed. If our data is to be representative of Velocity it must cover as many of these branches as possible. This is referred to as "exercising" the code to obtain a set amount of "code coverage".

To exercise code involves using as many features of a program as possible. If the application to be analysed does not include a suite of automated tests this might pose a problem, as exercising the code manually is both time consuming and error prone. Comparing results from two different versions of the same application would, for example, require using the exact same features in each version.

When dealing with open-source Java projects exercising the code manually is often not necessary, as many of them seem to have embraced automatic testing. This has perhaps been fuelled by initiatives such as the JUnit framework [GB04]. In the case of Velocity there are automatic unit-tests available.

However, automatic unit-tests are far from perfect. We need to be assured that they provide a satisfactory level of code coverage before we may claim that our study is representative of Velocity.

Deciding if an application has been properly exercised is nearly impossible without using special tools, as there are just too many variables to measure manually. Luckily, there are both commercial and open-source tools available for this job. The coverage tools output percentages showing how much code is covered by individual, or combinations of, unit-tests at the package, class and method levels. These tools can be used to check whether automated unit-test actually cover the required amount of source code and functionality.

Clover

We decided to use a coverage measurement tool called Clover. It is a tool written in Java by the Australian company Cortex. Clover was chosen both for its availability and maturity.

The process of achieving the desired level of code coverage started by running the Velocity unit-tests with Clover. Unfortunately, the initial level of code coverage was not satisfactory; only around 54% for Velocity version 1.2-rc1.

This lead us to examine the Velocity source code more closely. We sought

to determine whether the unit-tests were not good enough, or if the low coverage was due to some other cause. We found that there were four primary reasons for the low initial coverage.

- Dead code due to restructuring/refactoring.
- Unused exception handling code.
- Unused convenience methods (e.g., get/set methods for class variables).
- Presence of classes meant for interfacing Velocity with other tools and frameworks (e.g., Tomcat and Struts) .

In addition we did not want to include any of the classes related to the JUnit testing framework itself.

Clover employs a filter much like to one used in `mcalc`. So that whenever a class was found to have no coupling to the rest of Velocity it was filtered out of the coverage analysis. After some iterations of excluding classes, running Clover and reading source code the level of coverage was above 90%.

The filter configuration used by Clover was subsequently rewritten to conform with the format used by `mcalc`. The complete `filter.conf` file used in the analysis of Velocity can be found in Appendix C.5.

3.6 Technical Choices

Work on Jdissect started in January 2002. Since then we have created four different revisions of the system. However, a large amount of the source code has remained constant. The differences between the versions have mostly been due to alternative data storage solutions.

In the course of this work we have made a number of choices regarding the technology and methods used to implement Jdissect. This section contains a brief survey of the choices and the reasoning behind them.

3.6.1 Separation of Data Collection and Analysis

The first draft designs of Jdissect contained only the library component, and no separate tool for computing coupling. It seemed as if separating the processes of gathering and analysing data would only lead to unnecessary overhead and complications.

This idea was dismissed because it required data to be collected each time a program was analysed. Data collection is time consuming, especially if the analysis target is feature rich or if data must be collected from many consecutive versions of the same system.

On examining some actual Java applications we realised that most of them employ unit- or regression-tests which are executed in separate runtime instances of the Java VM. This means that data from multiple unit-tests must be merged in order to compute coupling measures for an entire application. Being able to merge data from multiple tests requires some form of intermediate storage.

These two factors led us to separate the data collection mechanism from the part of the program which performs data analysis.

3.6.2 Data Storage

There were several possibilities with regard to how Jdissect was to store data. The original idea was to use an Object Database Management System (ODBMS) supporting Object Query Language (OQL). Data storage could have been handled by the database, while calculation of the coupling measures could be performed using OQL instead of writing specialised C++ code.

We soon found it impossible to use an ODBMS, and proceeded with an attempt to use an ordinary Relational Database Management System (RDBMS) instead. Unfortunately, this approach turned out to be too slow for our requirements.

In the end we settled for a data storage solution based on flat text files. This provided benefits both with regard to performance and debugging.

ODBMS

Using an object database seemed like an elegant solution in theory. However, in practise we uncovered two major obstacles.

The first problem we encountered was that storing data in the ODBMS was slow. Starting a small Java program which does next to nothing will generate more than 10000 messages, load around 250 system classes containing over 1600 different methods and instantiate approximately 3500 objects. Each of these entities is represented by a separate object in the core data model. In theory it should be possible to store this amount of data in any database relatively quickly. The problem we found was at least partially due to the fact that many of the classes in the core data model reference each other. Storing this information, including the references between the different objects, while maintaining indexes and database integrity is very time consuming. In fact, even for rather small Java programs, the time required to store the core data model approached 15 minutes.

The second obstacle in using the ODBMS was that it did not implement the mechanisms we needed to use it efficiently. We believed, based on claims made by the database vendor, that the object database we selected conformed to and supported the entire OQL standard. This would have

enabled implementation of the dynamic coupling measures entirely in OQL. The reason for wanting to use OQL to implement the measures was two-fold: 1) Changing the measures if errors were found would be easy, and 2) new measures could be implemented rapidly.

However, implementing the calculation of the measures in an efficient manner would have meant sending *single queries* to the database, letting it handle traversal and selection from the core data model. Using single queries would have required the database to support selection from multiple tables; so-called "joins". However, the ODBMS we selected, based on claimed conformance to OQL, turned out to not support this operation. The database vendor advised using collection classes instead. Heeding this advice would have meant building the queries incrementally, selecting from one table at a time, and using external code (i.e., C++) to assemble the query results.

After some attempts to work around this limitation we decided against using an ODBMS.

RDBMS

Not entirely willing to give up on using a database we turned our attention to using a relational database management system (RDBMS).

We choose to experiment with MySQL, as it is generally considered quicker than other databases for single-user and non-concurrent scenarios.

Unfortunately, MySQL turned out to be another disappointment. While it performed significantly better than the object database, storing data generated by a small Java application took close to 4 minutes.

Flat-file Structure

In the end what seemed simplest and quickest was to implement data storage using a structure based on flat text files. This solution turned out to perform significantly better, as the entire execution history generated by a small program could be stored in under a second.

Additionally, the use of text files made debugging significantly easier since it was possible to manually inspect the stored data.

3.6.3 The Java Interface

Prior to implementing Jdissect we evaluated alternative methods for collecting data from a running program. We found that there were several important issues which should be considered.

1. It should be possible to reuse much of the Jdissect source code if we decided to analyse applications written in different programming languages.

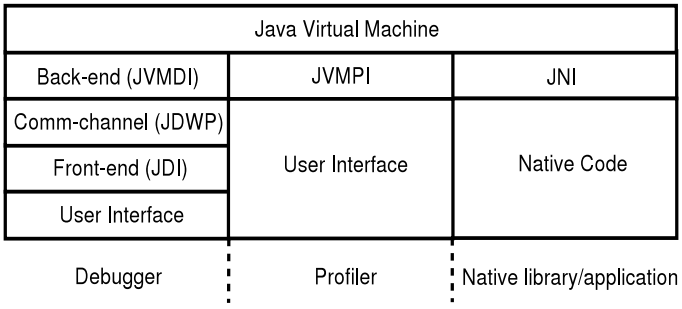

Figure 3.6: JPDA/JVMPI/JNI architecture

2. The interface to the running program should provide us with the information needed to populate the core model.

3. As data collection happens while the system is running, possibly interactively, speed and responsiveness are important.

The following section contains a short explanation of the various programming interfaces we could have used to collect run-time data from a Java application. Arguments for and against using the different interfaces are presented in turn.

Java Architecture

The Java Platform Debugger Architecture (JPDA) consists of three interfaces for use by debuggers.

The Java Debug Interface (JDI) is a high level Java API providing information useful for debuggers and similar systems which require access to the running state of a Virtual Machine. It defines information and requests at the user code level. It is a Java-only interface, and is recommended by Sun for both stand-alone debugging-tools and integration with developer environments. The JDI is the "highest" (e.g., most abstract) layer of the JPDA.

The Java Debug Wire Protocol (JDWP) defines the format used in information and requests transferred between the process being debugged (the JVM) and the debugger front end. Because of the JDWP it is possible to debug Java applications from remote computers.

The Java Virtual Machine Debug Interface (JVMDI) defines services which a VM can provide for debugging at a low level. It is a two way interface which enables clients to both monitor an application through events and querying it by using functions. The JVMDI is the lowest layer within the Java Platform Debugger Architecture.

The Java Virtual Machine Profiling Interface (JVMPI) is not yet part of the Java standard, but rather an experimental feature. Controlling and querying the JVMPI functions much like in the JVMDI. As the JVMPI is still experimental it does not yet have an interface from Java itself.

The Java Native Interface (JNI) is a standard programming interface for accessing native code from Java and for embedding the Java virtual machine into native applications. One of the goals of the interface is to enable application programmers to access the internals of the Java VM from native code on any given platform.

JDI

The JDI provides a high level Java interface to the internals of the Virtual Machine. Using it as the interface between Java and Jdissect requires Jdissect to be implemented entirely in Java. This is beneficial with respect to portability. If the JDI is used, Jdissect can be used on any platform where Java is deployed.

However, implementing Jdissect entirely in Java might involve compromises we are not prepared to accept. There are three issues which should be considered; efficiency, extensions of Jdissect and the availability of required information.

As discussed in Section 3.1.1 Java programs are byte code interpreted. In other words, execution of a Java program will in most instances be slightly slower than running similar programs compiled directly into native machine code. Implementing Jdissect entirely in Java might therefore incur performance penalties.

The class *Message* in the core model represents a method call. If the coupling measures are to be calculated correctly, this class needs to contain not only source/target method and object, but also the line number in the source method from which the target method is called. Unfortunately the information regarding line number of the dispatch from the source method is not easily obtainable from the JDI.

One of the goals we set for Jdissect is that it should be extensible. Implementing support for gathering data from other object-oriented programs should therefore be made as easy as possible. A pure Java implementation of Jdissect will retain a dependency on the Java VM, even if a new data harvesting mechanism was devised.

JavaCC

JavaCC is a completely different approach to gathering data. JavaCC reads a description of a programming language ('grammar') and generates a parser in Java. The parser blueprint must be modified to take various actions depending on the language constructs it encounters.

The reason why this approach is different from using any of the direct-interface strategies is that it requires source code instrumentation. The instrumentation process re-writes the source code of the application which is to be analysed. For example, every class must call a function registering it with the core model. Similarly, each method call must register itself. When the source code has been changed by the parser generated by JavaCC, the entire Java application must be recompiled.

There are two principal arguments against using this approach to collect data. Firstly, the instrumentation process requires access to the source code of the application which is to be analysed. Secondly, the functions inserted in the source code to register, for example, method calls with Jdissect will have to be implemented in the same language as the source code being analysed. In itself this is not a problem. The core functionality of Jdissect might still be implemented in a language other than Java, thereby avoiding possible performance penalties. However, this solution requires a bridge, or an adaptor layer, between Jdissect and the re-written source code. The adaptor itself must be implemented in the same language as the re-written source code, thereby possibly incurring any performance overhead associated with Java.

BCEL

The Byte Code Engineering Library (BCEL) is a Java based framework for modifying byte code files. In essence, BCEL can accomplish the same as JavaCC, but at a different level. While JavaCC can instrument source code, BCEL can be used to instrument compiled Java executables.

A solution using BCEL will possibly suffer the same problems as an implementation employing JavaCC with regard to using an adaptor layer between Java and Jdissect. Furthermore, the task of instrumenting Java classes at the byte code level was deemed too complex for use in Jdissect.

JVMPI/JVMDI

Considering the issues listen in at the beginning of Section 3.6.3, none of the data collection mechanisms presented so far seem ideal. They do not provide the information required to populate the core data model, are too complicated or suffer from possible performance issues.

There remains only three alternatives. Using the JVMPI, the JVMDI or JNI. However, none of these interfaces provide all the information required to populate Jdissects core data model by themselves. The solution is to use a combination of the alternatives.

These interfaces (JVMPI/JVMDI/JNI) are meant to be used directly with either C or C++. Thus, any concerns regarding performance can be set aside, as these languages ensure that the programming language (or rather

the compiled code) used to implement Jdissect is not an obstacle in achieving speed and responsiveness.

We choose to use C++ rather than C to implement Jdissect. This choice is motivated by the fact that the core data model is an object structure. Using C++ will make it relatively easy reuse most of the source code if we at some point decide to expand Jdissect to analyse programs written in other OO languages. That is, if the language we decide to analyse can interface with C or C++. Fortunately, C and C++ are both very common languages.

3.7 Summary

In this chapter we have described a tool which can be used to calculate the dynamic coupling measures proposed in Chapter 2. The tool is implemented in C++, and interfaces with the Java Virtual Machine by way of Java's profiling (JVMPI), debugging (JVMDI) and native (JNI) APIs. Data collected by `libjdissect.so` is stored on disk as a collection of text files. These files can be analysed using `mcalc`.

We have also described some of the steps taken to verify that the implementation of Jdissect functions according to the specifications laid out in Chapter 2.

In Section 3.5 we described how data was collected and analysed to gather data for the Velocity case study. The study itself, and its conclusions, can be found in Chapter 2.

References

[AN02] J. Arlow and I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Pearson Education Limited, Edinburgh Gate, Harlow CM20 2JE, 2002.

[ASF04a] The Apache Software Foundation. The Apache Ant project. http://ant.apache.org/, 2004.

[ASF04b] The Apache Software Foundation. The Apache Jakarta project. http://jakarta.apache.org/, 2004.

[ASF04c] The Apache Software Foundation. The Apache Velocity project. http://jakarta.apache.org/velocity/, 2004.

[GB04] E. Gamma and K. Beck. Junit - regression testing framework. http://www.junit.org/index.htm, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1994.

[HDFW03] T. Husted, C. Dumoulin, G. Franciscus, and D. Winterfeldt. *Struts in Action*. Manning, 2003.

[Knu97] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley Longman, 3rd. edition, 1997.

[Ste93] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.

Appendix A

Appendices to Chapter 2

A.1 Definition of the Size Measures

Some of the size measures in the text are frequently used in publications and available tools, and no definite source or author can be given for them.

Table A.1: Definition of the Size Measures

Name	**Definition**
NAI	The number of non-inherited attributes in a class
NAD	The number of inherited attributes in a class
NA	The total number of attributes in a class. NA = NAI + NAD
NMI	The number of methods implemented in a class (non-inherited or overriding methods)
NMD	The number of inherited methods in a class, not overridden
NM	The number of all methods (inherited, overriding, and non-inherited) methods of a class. NM = NMI + NMD
NMpub	The number of public methods implemented in a class.
NMnpub	The number of non-public (i.e., protected or private) methods implemented in a class.
NumPara	Number of parameters. The sum of the number of parameters of the methods implemented in a class.
CS1	The number of source lines of code in a class
CS2	The number of declarations and statements (semicolons) in a class

A.2 Informal Definitions of the Static Coupling Measures

Table A.2: Informal Definitions of the Static Coupling Measures

Name	Definition	Source
CBO	Coupling between object classes. According to the definition of this measure, a class is coupled to another, if methods of one class use methods or attributes of the other, or vice versa. CBO is then defined as the number of other classes to which a class is coupled. This includes inheritance-based coupling (coupling between classes related via inheritance).	[CK94]
CBO'	Same as CBO, except that inheritance-based coupling is not counted.	[CK91]
RFC	Response set for class. The response set of a class consists of the set M of methods of the class, and the set of methods directly or indirectly invoked by methods in M. In other words, the response set is the set of methods that can potentially be executed in response to a message received by an object of that class. RFC is the number of methods in the response set of the class.	[CK91]
RFC_1	Same as RFC, except that methods indirectly invoked by methods in M are not included in the response set.	[CK94]
MPC	Message passing coupling. The number of method invocations in a class.	[LH93]
DAC	Data abstraction coupling. The number of attributes in a class that have another class as their type.	[LH93]
DAC'	The number of different classes that are used as types of attributes in a class.	[LH93]
ICP	Information-flow-based coupling. The number of method invocations in a class, weighted by the number of parameters of the invoked methods.	[LLWW95]
IH-ICP	As ICP, but counts invocations of methods of ancestors of classes (i.e., inheritance- based coupling) only.	[LLWW95]
NIH-ICP	As ICP, but counts invocations to classes not related through inheritance.	[LLWW95]
PIM	Polymorphically invoked methods. The number of invocations of methods of a class c by other classes (regardless of the relationship between classes). Same as ICP, except that no weighting by the number of parameters is performed.	
PIM_EC	Export coupling version of PIM. The number of invocations of methods of a class c by other classes (regardless of the relationship between classes).	
ACAIC OCAIC DCAEC OCAEC ACMIC OCMIC DCMEC OCMEC AMAIC DMAIC AMMIC OMMIC DMMEC OMMEC	These coupling measures are counts of interactions between classes. The measures distinguish the relationship between classes (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction. The acronyms for the measures indicates what interactions are counted: The first or first two letters indicate the relationship (A: coupling to ancestor classes, D: Descendants, O: Others, i.e., none of the other relationships). The next two letters indicate the type of interaction: CA: There is a Class-Attribute interaction between classes c and d, if c has an attribute of type d. CM: There is a Class-Method interaction between classes c and d, if class c has a method with a parameter of type class d. MM: There is a Method-Method interaction between classes c and d, if c invokes a method of d, or if a method of class d is passed as parameter (function pointer) to a method of class c. The last two letters indicate the locus of impact: IC: Import coupling, the measure counts for a class c all interactions where c uses another class. EC: Export coupling: count interactions where class d is the used class.	[BDM97]

A.3 Descriptive Statistics

Table A.3: Descriptive Statistics

Variable	N	Mean	Median	Minimum	Maximum	Q1	Q3
IC_OC	136	6.95	1	0	108	0	6
IC_OM	136	9.59	2	0	144	0	7
IC_OD	136	10.93	2	0	182	0	9
EC_OC	136	6.95	3	0	79	0	7
EC_OM	136	9.59	4	0	101	0	11
EC_OD	136	10.93	4	0	117	0	12
IC_CC	136	5.21	1	0	108	0	5
IC_CM	136	6.93	1	0	144	0	7
IC_CD	136	8.69	1	0	182	0	9
EC_CC	136	5.21	2	0	64	0	5
EC_CM	136	6.93	3	0	138	0	5
EC_CD	136	8.69	3	0	221	0	6
CBO	136	4.13	2	0	43	1	5
CBO'	136	3.62	2	0	43	1	4
RFC_1	136	45.29	23	0	186	4	98
RFC_oo	136	290.90	31	0	792	4	718
MPC	136	6.26	2	0	116	0	8
PIM	136	14.90	3	0	126	0	28
PIM_EC	136	14.90	4	0	211	1	19
ICP	136	30.65	6	0	256	0	53
IH-ICP	136	3.84	0	0	174	0	2
NIH-ICP	136	26.81	6	0	256	0	43
DAC	136	0.47	0	0	9	0	1
DAC_	136	0.43	0	0	6	0	1
ACAIC	136	0.10	0	0	3	0	0
OCAIC	136	0.38	0	0	9	0	0
DCAEC	136	0.10	0	0	3	0	0
OCAEC	136	0.38	0	0	14	0	0
ACMIC	136	0.13	0	0	4	0	0
OCMIC	136	3.18	2	0	36	0	4
DCMEC	136	0.13	0	0	6	0	0
OCMEC	136	3.18	0	0	88	0	2
AMMIC	136	1.24	0	0	15	0	1
OMMIC	136	5.03	1	0	116	0	3
DMMEC	136	1.24	0	0	80	0	0
OMMEC	136	5.03	0	0	98	0	2
AMAIC	136	0.91	0	0	40	0	1
OMAIC	136	0.01	0	0	1	0	0
DMAEC	136	0.91	0	0	40	0	0
OMAEC	136	0.01	0	0	1	0	0
NA	136	9.65	6	0	133	1	10
NAI	136	3.59	1	0	68	0	4
NAD	136	6.06	0	0	107	0	10
NM	136	16.90	12	0	161	3	29
NMImp	136	9.12	4	0	161	2	8
NMD	136	7.78	0	0	36	0	24
NMpub	136	14.96	10	0	50	2	29
NMnpub	136	1.94	0	0	113	0	0
NumPara	136	10.31	6	0	146	2	9
CS1 (SLOC)	136	126.50	46	1	3766	25	98
CS2 (#semicolon)	136	56	15	0	1747	9	46

A.4 Principal Component Analysis for the Dynamic Coupling Measures

Table A.4: Descriptive Statistics

Variable	**PC1**	**PC2**	**PC3**	**PC4**
IC_OC	0.311	0.275	**0.892**	0.121
IC_OM	0.236	0.290	**0.918**	0.110
IC_OD	0.209	0.374	**0.897**	0.078
IC_CC	0.169	**0.909**	0.235	0.258
IC_CM	0.144	**0.912**	0.318	0.203
IC_CD	0.126	**0.912**	0.346	0.115
EC_OC	**0.911**	0.180	0.196	0.286
EC_OM	**0.884**	0.167	0.301	0.302
EC_OD	**0.855**	0.097	0.338	0.359
EC_CC	0.507	0.271	0.065	**0.804**
EC_CM	0.305	0.200	0.108	**0.923**
EC_CD	0.215	0.146	0.117	**0.956**

A.5 Principal Component Analysis for All Measures

Table A.5: Descriptive Statistics

Variable	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
IC_OC	0.381	-0.007	-0.144	-0.034	0.370	-0.071	**-0.786**	-0.021	-0.055	-0.034	-0.005
IC_OM	0.335	0.001	-0.097	-0.041	0.333	-0.079	**-0.829**	-0.014	-0.025	-0.047	0.033
IC_OD	0.428	-0.007	-0.097	-0.031	0.284	-0.064	**-0.819**	-0.002	-0.037	-0.037	0.030
EC_OC	0.315	-0.031	-0.310	-0.010	**0.778**	0.012	-0.319	0.002	-0.205	0.036	-0.031
EC_OM	0.215	-0.013	-0.287	-0.026	**0.827**	0.003	-0.364	-0.005	-0.083	0.107	0.004
EC_OD	0.163	0.010	-0.205	-0.038	**0.883**	-0.023	-0.335	-0.006	-0.033	0.114	0.017
IC_CC	**0.610**	0.112	-0.179	0.026	0.170	-0.014	**-0.551**	0.032	-0.337	0.092	0.185
IC_CM	**0.592**	0.091	-0.173	0.025	0.142	-0.013	**-0.613**	0.043	-0.313	0.078	0.179
IC_CD	**0.628**	0.061	-0.181	0.024	0.108	-0.001	**-0.621**	0.048	-0.257	0.072	0.152
EC_CC	0.054	**0.533**	-0.081	-0.075	**0.751**	-0.106	-0.049	0.061	-0.111	0.168	0.161
EC_CM	0.034	**0.682**	-0.019	-0.060	**0.615**	-0.155	-0.090	0.084	-0.102	0.245	0.114
EC_CD	0.017	**0.737**	0.009	-0.056	**0.552**	-0.174	-0.082	0.087	-0.092	0.232	0.077
CBO	0.069	0.378	-0.141	0.262	-0.057	**-0.766**	-0.030	-0.031	-0.079	0.101	0.228
CBO'	0.080	0.376	-0.060	0.270	-0.073	**-0.780**	-0.018	-0.028	-0.092	0.025	0.216
RFC_1	0.277	0.082	**-0.886**	-0.057	0.129	0.023	-0.204	-0.016	0.003	0.076	0.157
RFC	-0.001	0.112	**-0.819**	-0.117	0.111	0.103	-0.179	-0.006	-0.001	0.168	0.297
MPC	**0.781**	-0.041	-0.125	-0.014	0.089	0.028	-0.363	0.030	-0.232	0.085	0.342
PIM	**0.574**	0.168	-0.310	-0.067	0.027	0.041	-0.426	0.093	-0.036	0.244	0.484
PIM_EC	0.002	**0.602**	-0.091	0.086	0.480	-0.496	0.043	0.008	-0.105	0.137	0.149
ICP	0.436	0.244	-0.323	-0.063	0.010	0.021	-0.451	0.092	-0.176	0.200	**0.557**
IH-ICP	-0.020	**0.849**	-0.212	-0.011	0.005	-0.131	-0.029	0.101	-0.110	0.360	0.011
NIH-ICP	0.481	-0.044	-0.275	-0.064	0.009	0.070	-0.480	0.063	-0.152	0.087	**0.601**
DAC	0.451	0.286	0.070	-0.014	0.176	-0.113	-0.182	0.065	**-0.720**	0.269	0.075
DAC'	0.390	0.223	0.107	-0.046	0.216	-0.107	-0.114	0.061	**-0.723**	0.271	0.133
ACAIC	-0.013	0.250	0.024	-0.004	0.148	-0.070	0.041	-0.016	-0.140	**0.885**	0.006
OCAIC	**0.507**	0.212	0.068	-0.014	0.133	-0.095	-0.219	0.079	**-0.741**	-0.079	0.081
DCAEC	-0.049	0.250	0.097	**0.896**	-0.067	-0.090	0.037	-0.047	0.052	-0.031	-0.003
OCAEC	0.031	-0.065	0.073	-0.033	0.000	-0.801	-0.006	0.110	-0.036	0.084	-0.147
ACMIC	-0.004	0.491	0.007	-0.016	0.090	-0.088	0.055	0.004	-0.146	**0.793**	-0.018
OCMIC	0.272	0.099	-0.145	0.038	0.164	-0.371	0.031	-0.120	-0.101	-0.024	**0.665**
DCMEC	-0.038	0.015	0.079	**0.913**	-0.007	-0.027	0.038	-0.025	0.080	-0.033	0.024
OCMEC	0.177	-0.120	-0.093	**0.663**	-0.021	-0.416	-0.054	0.044	-0.211	0.055	-0.093
AMMIC	-0.077	0.037	-0.475	-0.019	0.270	0.055	-0.135	0.015	0.115	0.684	0.179
OMMIC	**0.810**	-0.049	-0.029	-0.010	0.034	0.017	-0.341	0.028	-0.259	-0.055	0.310
DMMEC	-0.007	**0.866**	0.005	0.352	-0.105	-0.215	0.030	0.041	-0.089	0.059	0.006
OMMEC	0.006	0.254	0.172	0.047	0.278	**-0.789**	-0.123	-0.028	0.036	-0.098	0.057
AMAIC	**0.553**	-0.112	-0.117	0.086	0.063	0.000	-0.527	0.009	-0.474	-0.061	-0.032
DMAEC	-0.062	0.452	0.039	-0.025	-0.101	-0.099	0.070	**0.691**	-0.149	-0.005	-0.093
NA	**0.778**	-0.055	-0.143	-0.030	0.156	0.029	-0.149	0.438	-0.155	-0.079	0.117
NAI	0.346	-0.017	0.177	-0.029	0.116	0.005	-0.055	**0.838**	0.017	0.017	0.029
NAD	**0.783**	-0.061	-0.300	-0.021	0.127	0.034	-0.157	0.024	-0.211	-0.113	0.132
NM	**0.751**	0.023	**-0.543**	0.043	0.200	-0.145	-0.167	-0.067	-0.144	-0.025	-0.007
NMI	**0.885**	0.050	-0.003	0.054	0.146	-0.234	-0.126	-0.034	-0.247	0.001	0.084
NMD	-0.111	-0.040	**-0.921**	-0.012	0.111	0.120	-0.087	-0.060	0.143	-0.043	-0.143
NMpub	0.264	0.073	**-0.826**	0.023	0.263	-0.217	0.010	-0.135	-0.058	-0.027	0.092
NMnpub	**0.912**	-0.044	0.033	0.043	0.033	0.004	-0.278	0.041	-0.167	-0.010	-0.111
NumPara	**0.737**	0.084	0.008	-0.071	0.104	-0.302	0.257	-0.063	0.229	-0.015	0.367
CS1	**0.967**	0.027	0.055	-0.014	0.047	-0.010	-0.150	0.071	0.050	0.026	-0.021
CS2	**0.961**	0.007	0.055	0.007	0.041	-0.021	-0.194	0.076	0.003	0.013	-0.041

References

[BDM97] L. C. Briand, P. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In *proc. 19th International Conference on Software Engineering (ICSE'97)*, pages 412–421, 1997.

[CK91] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming: Systems, Languages and Applications*, volume 26, pages 197–211. SIGPLAN Notices, Oct. 1991.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[LH93] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 1993.

[LLWW95] Y. S Lee, B. S. Liang, S.F. Wu, and F. J. Wang. Measuring coupling and cohesion of object-oriented programs based on information flow. In *proc. International Conference on Software Quality*, 1995.

Appendix B

Technical Details

This section contains more detailed descriptions of the algorithms and technology used in the development of Jdissect.

The first few sections are devoted to the relation between theoretical sets, like ME and IV, and their implementation. This is followed by explanations of how various Java programming interfaces are used, and how Jdissect saves/loads its core data structure.

B.1 Set Implementation

This section explains how each set is maintained *within* the *SetContainer* class. The structures and methods discussed are also used to maintain the relations between the different classes in the core model.

The Standard Template Library

Jdissect is implemented using C++. The reasons behind the choice of programming language are examined in-depth in section 3.6. But in order to explain the data structures used to implement sets and relationships in the core mode we have to briefly introduce the STL (Standard Template Library), which is closely related to C++.

The STL was developed at Hewlett-Packard Labs by Alexander Stepanov and Meng Lee [SL94]. Their work was based on earlier papers by Stepanov and Musser. For an historical overview and complete reference, see [PSLM01]. The STL is now part of the ANSI/ISO C++ standard. But the standard itself is not suitable as documentation. [Str97] is both readable and follows the ISO standard.

The standard template library is meant to make developers more productive by providing standardised components. This saves time on both development and testing. Such standardisation also makes programs easier

to read for someone who did not develop them, as everyone knows the meaning of names and functions in the standard.

Nearly all STL code falls into one of three categories; algorithms, iterators and containers [PSLM01, MDS01]. In this section we are mostly concerned with describing containers.

The STL `set` Class

Implementing efficient object containers from scratch in C++ usually requires a lot of effort. The STL provides a host of different container structures. One of these structures is called 'set'. It resembles the mathematical definition of a set, and supports the use of operations like union, intersection and difference. It is ideal for use in implementing both inter-class relations and the containers in *SetContainer*. Worst case performance for looking up an object contained in a set is $O(\log N)$. The speed is required, as the container will often be used to hold very large amounts of data.

A `set<Key, Compare>` stores unique elements of type `Key`. Elements in the set are ordered using the functor (function object) `Compare`, which must induce a *strict weak ordering* on the elements. For a precise definition of this requirement see [MDS01, p. 411].

Using sets might seem daunting at first, especially the rules governing insertion and removal of elements. Meyers provides more elaborate insights into both sets and the STL in [Mey01].

The `Compare` functor is used both for ordering elements and comparisons during lookup. This template argument has a default value of `less<Key>`. This function object uses the less-than ('<') operator to compare two elements. If the two elements being compared are values a simple numerical comparison is made. But if the elements are objects instantiated from classes that have had their less-than operator (`operator<()`) overridden, this function is used instead. The less-than operator is often to used to implement more complicated comparisons based on various class attributes.

In the Jdissect core data model all references to other objects are pointers. Therefore all instances of `set` use a modified version of the `less<Key>` functor called `ptr_less`.

Listing B.1: `ptr_less` definition

```
1 template <typename T>
2 struct ptr_less : public binary_function<T, T, bool> {
3     bool operator() (const T& p1, const T& p2) const {
4         return (*p1) < (*p2);
5     }
6 };
```

The only difference between the default `less<Key>` and the definition in listing B.1 is that the latter dereferences its two arguments before attempting to apply the less-than (<) operator. If we had omitted dereferencing the

functor arguments the pointer values would have been compared instead of the object contents. This would leave elements ordered according to location in memory, instead of according to element content.

All the sets used in the core model, and in *SetContainer*, are based on STL sets. To create a more consistent programming interface we created type-definitions (`typedef`'s) for each kind of element. The set for storing instances of the class *Message* is defined in listing B.2. Similar type definitions exists for all classes in the core data model.

Listing B.2: `message_set` type-definition

```
typedef set<Message*, ptr_less<Message*> > message_set;
```

Uniqueness

This preoccupation with describing the STL set stems from the need for a container class which stores unique elements. Uniqueness is required to prevent duplicate data from being stored in *SetContainer* and in the inter-class relationships in the core model. Failure to control the set container would mean invalidating the theoretical definitions from chapter 2, and ultimately jeopardising the integrity of the coupling measurements.

At first glance the core model classes look like little more than collections of data with get/set methods. They do however share one distinguishing feature; the less-than operator, $<$, is overloaded in each class. As previously described this operator controls the criteria used by sets to determine if an object is unique or not. If we examine listing B.1 again this operator is applied to the functor arguments in line 4.

Table B.1 shows which attributes are compared in the overloaded less-than operator of each core model class. In most cases comparing the attributes is simply a matter of once again employing the less-than operator. However, this operation should not be attempted directly on pointer attributes since they might be set to `NULL`. In this case we first check both sides of the expression before attempting a direct comparison.

All overloaded less-than operators in the core model return 0, i.e., 'false', if the objects compared have only equal values. This fulfils the *strict weak ordering* criteria we described previously (see also [Mey01, p. 92].

B.2 MethodInvocation

To describe *MethodInvocation* we have to explain the relationship between ME and IV. The set ME contains all instances of the *Message* class, and is directly based on all function calls that occur in a Java program. IV, on the other hand, is what we refer to as a derived set. In other words; it is created from something else. Instances of *MethodInvocation* in IV are created based on the *Message*s in ME.

Table B.1: Comparisons made in overloaded less-than operator

Class	**Uniqueness criteria**	**Notes**
Attribute	type, name	Inherited by StaticAttribute
Parameter	type, name	Inherited by InputParamter, OutputParamter and InOutParameter
Object	oid, instanceOf	
TestCase	tid, name	
Class	cid, name	
Method	signature	
Message	location, isReflexive(), systemImport, exportObject, importObject, exportMethod, importMethod, threadName	
MethodInvocation	location, exportClass, importClass, exportMethod, importMethod, isReflexive()	

In essence, the difference between a message and a method invocation is that the invocation refers to class instead of object. But not simply the class from which an object is instantiated. The invocation instead refers to the classes in which the methods used were first defined or last overridden.

The concept is perhaps easier to understand based on the algorithm used to derive IV from ME.

Deriving IV from ME

In chapter 2, the relationship between ME and IV is defined formally using a set theory and first order logic. The expression is shown again in equation B.1 for reference. Although the consistency rule looks complicated, the corresponding algorithm is far from incomprehensible.

$$\begin{aligned}(\exists(o_1,c_1),(o_2,c_2)\in R_{OC})(\exists l\in\mathbb{N})(o_1,m_1,l,o_2,m_2)\in ME \Rightarrow \\ (\exists c_1'\in A(c_1)\cup\{c_1\}, c_2'\in A(c_2)\cup\{c_2\}) \\ ((m_1,c_1')\in R_{MC}\wedge((\forall c_1''\in A(c_1)-\{c_1'\})(m_1,c_1'')\in R_{MC}\Rightarrow c_1''\in A(c_1')))\wedge \\ ((m_2,c_2')\in R_{MC}\wedge((\forall c_2''\in A(c_2)-\{c_2'\})(m_2,c_2'')\in R_{MC}\Rightarrow c_2''\in A(c_2')))\wedge \\ (m_1,c_1',m_2,c_2')\in IV \quad \text{(B.1)}\end{aligned}$$

As explained in B.2 the difference between instances of *MethodInvocation* and *Message* is mainly that messages refer to caller/callee objects and methods , while method invocations refer to the classes defining the methods used.

In listing B.3 `ME` and `IV` refers to the sets of messages and method invocations, respectively. The `FOR-EACH` statements are loops which iterate over the contents of a set. `RecursiveFindClass` is a function which accepts *Class* and *Method* as parameters, and returns the *Class* in which the *Method* is defined.

All 'sanity' checks, such as checking if the call to `RecursiveFindClass` returns `NULL`, have been removed from the pseudo-code for brevity and clarity.

Listing B.3: Algorithm for deriving IV from ME, expressed in pseudo-code

```
1  NEW IV
2
3  FOR-EACH Message IN ME
4  BEGIN
5      NEW MethodInvocation
6      MethodInvocation.location = Message.location;
7      MethodInvocation.importMethod = Message.importMethod;
8      MethodInvocation.exportMethod = Message.exportMethod;
9      MethodInvocation.importClass =
10         RecursiveFindClass(Message.importObject.instanceOf,
11                            Message.importMethod);
12     MethodInvocation.exportClass =
13         RecursiveFindClass(Message.exportObject.instanceOf,
14                            Message.exportMethod);
15
16     ADD MethodInvocation TO IV
17 END
18
19 Class RecursiveFindClass(Class c, Method m)
20 BEGIN
21     IF( m IN c.methods )
22         return c;
23     FOR-EACH a IN c.Ancestors
24     BEGIN
25         foundClass = RecursiveFindClass(a, m);
26         IF( foundClass != NULL )
27             RETURN foundClass;
28     END
29     RETURN NULL;
30 END
```

Line 1 sees the creation of a new set, IV, to hold *MethodInvocations*. Iteration over the set ME starts in line 3. The contents of the loop mainly copies the contents of each *Message* into a corresponding *MethodInvocation*,

except the attributes `importObject` and `exportObject`.

The two import/export class attributes are set in lines 8 through 13 by calling `RecursiveFindClass`. This function first searches the class it received as parameter `c` for a definition of the method `m`. If `m` is not defined in this class, the function continues to search all ancestors of `c`.

For example, if a class overrides its parents version of a method, or is itself the first class in the inheritance hierarchy to implement it, a reference to this class is copied to the *MethodInvocation*.

However, if a parent class contains the declaration, a reference to the parent is copied instead. The result is that the *MethodInvocation* contains references to the same two *Methods*, and the same *location*, as the *Message* it is created from. But instead of referring to specific objects involved in a function call, it has references to the classes declaring the source and target methods.

B.3 The Profiling Interface - JVMPI

Jdissect has to interface with a running instance of the Java VM to collect data. `libjdissect.so` starts the process by registering a callback function with the VM. All communication from the VM to the library is event-driven, while the library controls the VM using function calls.

The Java Virtual Machine Profiling Interface (JVMPI) was originally created by Sun as a programming interface for developing profilers. Although it does not seem to be part of the Java standard, most implementations now support it.

Nearly all the information gathered by `libjdissect.so` originates in the profiling interface. The remainder is collected from the Java Virtual Machine Debug Interface (JVMDI) and the Java Native Interface (JNI), which are discussed in section B.4.

Data Types

Interfacing with Java requires some way of uniquely identifying the various parts of an object-oriented system. JVMPI refers to entities such as classes, methods, threads and objects by using different identities, or IDs. Any ID has both defining and undefining events, and is valid for the period of time between them. After an undefining event the ID can be reused for other purposes.

Data types like `jobjectID` and `jmethodID` are defined a C header file which holds definitions required to interface with the JVMPI (`jvmpi.h`). These two types are pointers to structures used within the VM. The structures themselves are of no interest to us. But the way in which they are declared means that these identity types are actually memory pointers. This

Table B.2: JVMPI identity types

Identity name	**Data type**	**Defining event**	**Undefining event**
thread ID	JNIEnv *	thread start	thread end
object ID	jobjectID	object alloc	object free, object move, and arena delete
class ID	jobjectID	class load	class unload and object move
method ID	jmethodID	defining class load	defining class unload
arena ID	jint	arena new	arena delete
JNI global ref ID	jobject	global ref alloc	global ref free

might however be subject to change, as the Java standard does not prevent anyone from implementing them differently.

The identities described in table B.2 are not used inside Jdissect, other than in the interface with JVMPI (everything in the *JvmpiInterface.cpp* file). There are two reasons for this decision.

- We can never be certain of how the various IDs are implemented in different Java versions. Their size and type are subject to change depending on VM version and the machine architectures definition of pointers. This can lead to unexpected behaviour, even between different versions supplied by the same Java vendor.

- Using the identities defined in the JVMPI would make Jdissect dependent on these types, even if another interface for gathering data was implemented later.

Instead the core model declares IDs that are of type `long`. Values are allocated to instances of the core model classes as they are instantiated.

Some of the data types used are specific to the JVMPI, while others are also found in the JNI (Java Native Interface). More information on these types and their uses can be found in [Lia99] and Sun's web-based documentation of the Java APIs.

Events

One of the first actions of `libjdissect.so` is to register a call-back function with JVMPI. The library proceeds by enabling all the events that it needs to monitor.

There is a total of 37 different events in the JVMPI. Jdissect only needs to monitor 13 of these. In general, the unused events provide even more detailed information on the run-time state of the virtual machine. But these detailed events are not required to populate the core model. For a list of enabled events and their function see appendix C.6.

Each event is accompanied by information pertaining to it. A class-load event will for example contain identity, class name, source file name, declared methods and attributes.

Because information is not retransmitted by the JVMPI it is important to register events when they occur. It is possible to request retransmission of certain information, but this quickly becomes inefficient. Furthermore, requesting information only works for defining events such as class load, thread start and object allocation (see table B.2). Jdissect therefore caches all information it receives, and never request that events are re-sent.

B.4 The Debug and Native Interfaces

In this section we will look at some of the differences between available and needed information in the various Java VM interface APIs.

Unfortunately the JVMPI does not provide all the information needed to populate the core model. In particular there are two areas where we have to resort to other APIs and methods in order to fulfil the theoretical requirements described in chapter 2.

Close examination of the xx_xD (Dynamic Message) measures reveal that they require some way of differentiating between multiple calls from one distinct method to another. This requires a unique number showing the line in the source code from which the method call originates. We refer to this number as the 'location' of the method call.

The location attribute can be seen as the lowest level of information granularity in messages. It is used both in calculating coupling, and in checking if messages are unique during updates to the core model.

This information is available from the JVMPI. However, the Java Virtual Machine Debug Interface (JVMDI) has functionality which can be used to obtain the stack frame location of each method call. This provides the needed location reference, and makes it possible to calculate the xx_xD measures.

In addition, each *Class* requires a reference to its immediate ancestor. The set of ancestor classes, or super-classes, is used when deriving the set of method invocations, IV, from the set of messages, ME.

Ancestor information can only be obtained though the Java Native Interface (JNI) [Lia99]. But only by using data initially received from the JVMDI.

Method Call Locations

It is easy to understand the reasoning behind most attributes in the core model class *Message*. The attribute *location* is an exception. The significance of *location* is best illustrated by an example.

Listing B.4: The use of *location*

```
1  class classA {
2      void methodA() {}
3  }
4
5  class classB {
6      void methodB() {
7          classA obj_a = new classA();
8          obj_a.methodA();
9          obj_a.methodA();
10         for(int i=0; i < 10; i++)
11             obj_a.methodA();
12     }
13 }
```

Listing B.4 shows two classes, `classA` and `classB`, and their methods. The problem lies in determining which method calls should be considered unique messages, and consequently added to the set ME. As we will see, the solution is closely linked to the location attribute.

The first call to `methodA` happens on line 8. As there have been no previous calls to this method from `methodB` it should be registered as a message with the core model. Jdissect stores the combination $(classB, methodB, 8, classA, methodA)$ as a *Message*.

Line 9 contains a second call from `methodB` to `methodA`. This method call is equal to the previous one in all but the line number. Even if the methods involved are the same, the line number is different. So the combination $(classB, methodB, 9, classA, methodA)$ is also stored as a unique *Message*.

What happens to the ten method calls inside the loop on line 11 is clearly interesting. The first time line 11 executes ($i = 0$) this method call is registered, because the combination $(classB, methodB, 11, classA, methodA)$ is not already in ME. However, the method calls in subsequent iterations of the loop ($i = 1 \cdots 9$) are not added to ME, as the combination is already present.

Obtaining *Location*

There is no way of obtaining any information regarding line numbers, from the profiling interface (JVMPI). However, the low-level debugging interface (JVMDI) has functionality which can be employed.

Listing B.5: JVMDI functions used to obtain unique origin of method dispatch

```
jvmdiError
  GetCallerFrame(jframeID called, jframeID *framePtr);

jvmdiError
  GetFrameLocation(jframeID frame, jclass *classPtr,
                   jmethodID *methodPtr, jlocation *locationPtr)
```

The two methods shown in listing B.5 do not provide the source line of the method dispatch. But the value obtained using `GetFrameLocation` is just as good. It provides the location of the instruction that is currently executing at the time of method dispatch.

If we use these functions on the example code in listing B.4 they provide us with the needed information. The stack frame locations reported for the invocations on line 8, 9 and 10 are all different. While subsequent iterations of line 10 all end up with the same location as the first iteration.

Problems with the Profiling Interface

Some features of the JVMDI interface function much like the JVMPI. Most significantly it makes use of events to signal state changes within the VM. There are also, however, substantial differences between the two interfaces.

While the profiling interface makes use of data-types like `jobjectID` and `jmethodID` to identify classes, objects and methods, the data types of the debugging interface are similar to those used by the Java Native Interface (JNI). JNI refers to objects and classes as having 'handles' of type `jobject` and `jclass`. Most information is gathered from the profiling interface, so we need a method of linking information from the two other programming interfaces to it. From listing B.5 it is possible to deduce that the easiest way of obtaining these handles is by translating the JVMPI identities.

While there exists two functions (see listing B.6) in the JVMPI specification which are supposed to translate one identity type to the other, these functions are unusable in practise.

Listing B.6: Unusable functions from the JVMPI

```
jobject (*jobjectID2jobject)(jobjectID jid);
jobjectID (*jobject2jobjectID)(jobject j);
```

The JVMPI specification states that the functions in listing B.6 are experimental and could be removed from future versions of the interface. It also warns that use of these functions is unsafe and can lead to unreliable and unstable profilers. Experiments in using them resulted in segmentation violations and subsequent program crash.

Mapping between the Profiling and Debug Interfaces

There is however another method which can be used to translate identities into handles. The two interfaces have many similar events. Fortunately,

there exists an event called `JVMDI_EVENT_METHOD_ENTRY` in the JVMDI that more or less corresponds to the JVMPI event `JVMPI_EVENT_METHOD_ENTRY2`. Both events are triggered on execution of a new method. The difference between the events is that the JVMDI version provides us with data which contains identifiers of type `jclass` and `jmethodID`.

The listings below (B.7 and B.8) show that the two method identifiers are directly comparable, as they have the exact same type. While the `jclass` handle in the event enables us to call JVMDI and JNI functions.

Listing B.7: Information from `JVMDI_EVENT_METHOD_ENTRY`

```
typedef struct {
    jthread thread;
    jclass clazz;
    jmethodID method;
    jframeID frame;
} JVMDI_frame_event_data;
```

Listing B.8: Information from `JVMPI_EVENT_METHOD_ENTRY2`

```
struct {
     jmethodID method_id;
     jobjectID obj_id;
} method_entry2;
```

The only problem is that the events detailing method calls are received from two different interfaces to the Java Virtual Machine. One is caught by the event handling (call-back) function set up for the JVMPI and the other by the function for JVMDI events.

To complicate things even further there is nothing in the Java documentation indicating which of these events are sent first. In practise this means that different Java vendors are free to implement them in whichever way suits them. In fact, if the vendors want to, their implementations might send a batch of profiling events before any debugging events are sent.

Jdissect solves this problem by creating a stack of method call information in the class `JvmpiController`. Entries are created based on either JVMPI or JVMDI events. Subsequent events describing method calls already existing on the stack are used to update the information. Entries are only removed from the stack, and added to the set of messages, ME, after execution of a method completes (the JVMPI event `JVMPI_EVENT_METHOD_EXIT`).

Ancestors

The core model represents each class in a running system with an object of type *Class*. In addition to implemented methods and attributes, each *Class* contains a reference to its superclass.

Knowledge of ancestor class is vital to the process of deriving the set of method invocations (IV), from the messages in ME.

JVMPI has no functionality which can be used to retrieve information about ancestors, and neither has JVMDI. But the Native Interface (JNI) contains a relevant function, [Lia99].

Listing B.9: JNI function to retrieve superclass

```
jclass (JNICALL *GetSuperclass)(JNIEnv *env, jclass sub);
```

The function shown in listing B.9 requires two arguments; a pointer to the native Java environment (`JNIEnv*`), and a `jclass` handle to the class in question. Any event from either the JVMPI or JVMDI interface provides the needed `JNIEnv` pointer. But only the debugging interface provides the needed `jclass` parameter.

This very much resembles the problem related to obtaining method call location information. The only difference is that the needed function is part of the JNI, instead of the JVMDI. Obtaining the appropriate handlers can still be accomplished by using JVMDI events. The class load event of the JVMPI is mirrored in the debugging API by an event called `JVMDI_EVENT_CLASS_PREPARE`.

There is one major difference between the code used to obtain ancestor information and the code used in registering method call location. Experiments showed that the `JVMPI_EVENT_CLASS_LOAD` event is always triggered before `JVMDI_EVENT_CLASS_PREPARE`. The event handling code for the JVMDI event relies on this, and expects class information to be present in the core model when it is triggered. There are no caching or mechanisms similar to that used to synchronise information from the two method entry events. Even though this strategy works in practise, the fact that there is no synchronisation mechanism is an oversight which should be remedied.

B.5 Threads and Locking issues

Java programs are always multi-threaded. Even programs that never explicitly use threads cause 6 threads to be created (using IBM's Java version 1.4.0, build cxia32140-20020917a). The threads are used for such tasks as garbage collection, reference handling and program execution.

Both the JVMPI and JVMDI run in the same process as the Java VM. Events from both APIs are sent in the context of their originating threads. As, for example, garbage collection and program execution happen in different threads Jdissect needs a locking strategy to prevent threads from attempting to modify shared data at the same time. Failure to lock common data structures during updates can result in deadlocks and data corruption [OW99, Lia99].

Locking Strategy

Jdissect deals with synchronisation in a very simple manner. Whenever an event is received, the call-back function in *JvmpiInterface* obtains a lock in order to ensure consistent updates. This prevents concurrent updates by other threads until the lock is released. While the call-back function has the lock it updates the core model through intermediary functions in *JvmpiController*.

The class *Lock* is responsible for synchronised data access. It can use one of two possible strategies.

- **Raw monitors** - Raw monitors are similar to Java monitors (i.e., using the Java `synchronised` keyword). The difference is that raw monitors are not associated with Java objects.
- **Pthreads** - POSIX threads [But97]. Provides lower-level locking than raw monitors. The exact level at which locking occurs depend on the operating system and the Pthread implementation.

Which locking strategy is used depends on the compile-time flag `USE_RAW_MONITOR`. If the flag is set, raw monitor locking will be used.

Choosing the correct locking strategy depends on the Java Virtual Machines thread implementation and the operating system. It seems like raw monitor locks mostly work under Linux. While Pthread locks should be used in a Sun Solaris environment.

Updates during Garbage Collection

The locking scheme is enforced for all event-handler code in Jdissect, with one notable exception related to garbage collection (GC).

During GC Java is in 'thread-suspended mode'. This means that there are some special restrictions on what the interfacing program is allowed to do. In particular, memory allocation is not allowed as it might block the current Java thread. Consequently, all GC related information has to be cached in pre-allocated buffers and merged into the core model once GC ends.

To prevent any of the non-GC related events from performing normal updates in thread-suspended mode the `JVMPI_EVENT_GC_START` event makes Jdissect obtain a lock that is not released until garbage collection has been completed. Only two events are possible during this period. Object move (`JVMPI_EVENT_OBJECT_MOVE`), and object free (`JVMPI_EVENT_OBJECT_FREE`).

Both events operate on data structures (*ThreadSuspendCache*) which are allocated when the VM first signals that garbage collection is about to start. The structures consist of large buffers that can be filled with descriptions of which objects are moved or deleted during GC. When the VM signals that

GC is about to end the lock is released and the buffered data is transferred to the core model.

Concluding remarks on Locking

While the strategy of locking the entire core model each time an event is processed can be time consuming it is very simple to implement. Informal tests concluded that the performance penalty is not great enough to warrant using 'thread local storage' or fine grained locking mechanisms aimed at locking only small parts of the core model at a time. This is at least partially due to the complex relationships between classes in the core model.

However, if Jdissect is ever to be used for heavily threaded applications it might be a good idea to re-write the locking mechanism used during method calls to take advantage of 'thread local storage'.

B.6 Storing Data

Transferring data from `libjdissect.so` to `mcalc` requires intermediate storage on disk. The process starts when the Java application being analysed terminates, and the library receives the `JVMPI_EVENT_JVM_SHUTDOWN` event.

There are in essence two different aspects of the core model that must be stored.

- Set contents, for example, the set of all messages, ME.
- Relations between sets, for example from messages to objects and methods.

Visitor Pattern

The algorithm used to store data is based on the Visitor pattern described in [GHJV94, p. 331]. Visitor decouples operations on a complex object structure from the structure itself. Thereby making it possible to implement new operations on the structure without changing its implementation.

Originally we envisaged applying the Visitor pattern to a wide range of operations in Jdissect. In addition to storing data, both measurement calculation and rebuilding the model from stored information were candidates for using the Visitor pattern. Data storage was the first of these operations to be implemented. As work progressed on the other two we found the pattern unsuitable in their cases. But the Visitor implementation remains in the storage algorithm.

Our implementation of the Visitor pattern contains a slight modification related to delegating responsibility for traversing the object structure. Responsibility for structure traversal can be implemented in one of three different places. In the object structure, in an iterator or in the visitor itself.

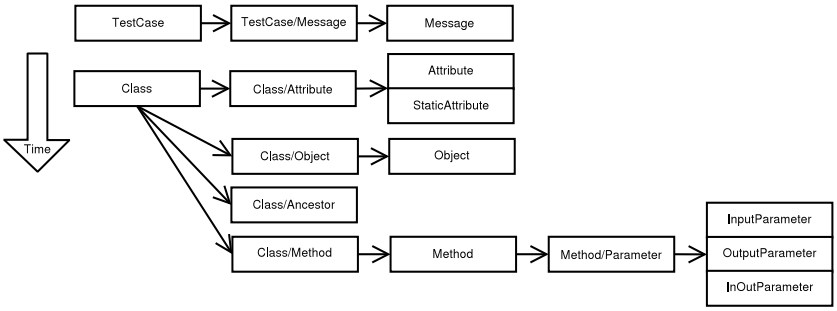

Figure B.1: Visitor traversal order

The first two alternatives are similar in that they tend to force traversal in a specific order [GHJV94, p. 339]. In our case, where we contemplated using the visitor pattern for many different tasks, this was unacceptable. There was no way of knowing up-front if the traversal order used when storing data would lend itself well to the tasks of calculating measurements or rebuilding the model from stored data.

Traversal order is therefore controlled from the visitor implementation. This modification is described in [GHJV94, p. 339], albeit with a warning that it will lead to duplicating the traversal algorithm in each concrete visitor type for each aggregate element in the structure.

Storage Order

Figure B.1 shows the order in which the core model is traversed and stored to disk. The process is split into two parts. The first starts by saving *TestCase*, while the second begins with *Class*.

Boxes containing two names, for example *TestCase*/*Message*, represents storage of object relations.

File Structure

`libjdissect.so` stores data in 16 different files. 10 of them contain data from the sets of objects in the core model, while the remaining 6 files store relationships between classes. A complete description of the various files and their formats can be found in appendix C.7.

The convention used to name the files is straightforward. Any filename that does not contain an underscore character ('_') stores a set of objects from the core model. Any file name containing an underscore stores the relation from the class named before the underscore to the one named after it.

Messages are a special case in that they do not contain sets of references to other core model classes. They refer to specific instances instead. As a

result, there are no files describing relations to or from the set of messages. All information regarding relations between instances of *Message* and the contents of other sets is stored directly in the file `message`.

B.7 Reading Data

In section 3.3.1 we explained how data is often collected from more than one unit- or regression-test, each executed in its own Java session. Data for each such test is always stored in separate directories.

Sometimes it might be interesting to analyse each data-set in turn. But we are usually interested in the result of summing up the measurements and analysing an entire application.

This requires some method of combining an unknown number of data-sets from different Java VM sessions pertaining to one application.

Reading Data - Naive Solution

An obvious solution would be merging the data stored in each Java VM session into one large core model. In some respects this would create an exact representation of what happens in the different unit-tests. But if we examine the data from the perspective of the VM there is a serious problem inherent in this approach.

As long as data has been collected from only one application nearly all the classes in the core model will remain constant between Java sessions. In fact, this observation is true for all sets in the core model except messages (ME) and objects (O). For example, the contents of the set of methods (M) will remain close to identical amongst the sessions.

ME and O are problematic because their content is closely tied to the *execution* of an application, rather than being tied to the *application* itself. As we will see, attempting to merge the contents of either set is infeasible.

An object within the Java VM is always unique, distinguished from other instances only by its allocated identity. When merging data there is no information that allows us to determine if two Java objects were created as a result of the exact same code-paths and conditions, so there is no way of comparing instances of *Object* across sessions. Consequently, the contents of O can only be concatenated, and never truly merged. This means that if we attempt to combine data from two sessions each containing 100 *Object* instances the resulting set O will have a cardinality of 200.

Because of the uniqueness criteria used in *Message* (see table B.1) the inability to properly merge *Object* data has the unfortunate side-effect of making each *Message* unique across sessions as well. Considering the many bi-directional references between *Message* and other classes in the core model it is evident that this solution will use far too much memory.

We attempted to merge data generated by 17 unit-tests from the Velocity application in an early experiment to see how large the ensuing in-memory representation of the core model would become. This approach rapidly filled 1 GB of memory and caused `mcalc` to crash with an 'out of memory' error.

Reading Data - Solution

If we examine the definitions of the measures (see Chapter 2) it is clear that none of them ever count distinct objects. On the contrary, all the measurements filter out unique *Object* instances and examine the class from which they were instantiated instead. For example, the most detailed measure uniqueness clause can be found in the xx_xD measurements, which use the criteria (c, m, l, c', m').

This opens the possibility of only importing objects which do not already exists in the merged core model. In effect this only performs some of the filtering that would normally occur at an earlier stage.

This strategy for merging *Object* data solves the problem of duplicate messages gracefully as well. The result of this approach is actually a merged core model which is smaller than we could ever have hoped for if the naive solution worked.

DataFileController

The *DataFileController* class is another subclass of *Controller*. In essence it performs the same task as *JvmpiController*, in that it gathers and forwards data to an instance of *ModelBuilder*. But while *JvmpiController* receives data from the Java VM, *DataFileController* opens a set of files and reads data saved by the Jdissect library.

DataMerger

When the `mcalc` program starts it first creates an instance of the *DataMerger* class. This, in turn, instantiates *DataFileController* which proceeds by reading and re-assembling the saved structure using *ModelBuilder*. In most cases there are saved data from several unit-tests which must be merged into a common structure.

The core models are reassembled by instances of *ModelBuilder* and *DataFileController*. After each load operation *DataMerger* performs the selection of which data is to be merged , while the process of moving data from one core model representation to another is handled by an instance of *ModelBuilder*.

In other words, the merge operation uses two instances of the builder class. One for assembling read data, and one for the merging process.

B.8 Implementing the Measures

We shall now examine how the theoretical definitions of measures from Chapter 2 can be decomposed in order to see their differences more clearly. An efficient implementation will necessarily be based on these observations.

We will then proceed by explaining some key concepts and then show how the theoretical measure definitions are implemented.

From Theory to Implementation

The definitions of the twelve different coupling measures are formally presented in Chapter 2. However, while those definitions are very precise in a theoretical sense it might be easier to understand how they can be implemented from a more practical or technical perspective.

There are three key differences between the various coupling measurements. The following list is an attempt at breaking down the variations into smaller, more manageable parts. The implementation of the measurements is based on these observations.

1. Direction - import or export coupling.
   (a) Import coupling - calls from a class to other classes. The importing class makes use of functionality defined elsewhere.
   (b) Export coupling - calls from other classes to a class. The exporting class is called from outside its own set of functions.
2. Entity of measurement - object or class level.
   (a) ME - the set of all messages. A message is a method call from a object/method/line tuple to a object/method tuple.
   (b) IV - the set of all method invocations. It is derived from ME, but is different in that elements refer to class instead of object. In the process of creating IV class is determined based on where the relevant methods are implemented, instead of being based simply on which class an object is instantiated from.
3. Strength - which can also be called the "set uniqueness criteria". Each actual measure is the cardinality of a set. Each set has its own uniqueness criteria. For example, $IC\_OD(c)$ and $IC\_CD(c)$ have uniqueness defined by (m, c, l, m', c'). m and c designates importing method and class, while l is the line in the importing method source code where the call originates. Target method and class is represented by m' and c'. This naming convention is consistently used for all definitions. Each type of criteria is the same for two measure types.

In addition to the differences between the measures, there is one criteria they all have in common.

- Non-reflexivity - all the measures specify $c \neq c'$. In practise this means that source and target class can never be equal. In other words, if a class calls its own methods it is not counted as coupling.

Functors and Function Object Composition

Functors are essentially classes containing an overloaded function operator (`operator()`). This makes it possible to use an object as a function call with state variables.

One of the techniques used in the implementation of the measure classes is what is known as function object composition [VJ02]. This technique makes it possible to compose several functors into an expression using templates. The fact that the composition relies on templates means that the expression is static. No time is therefore wasted on dynamic class type lookup during execution.

Example Definitions

Most of the code dealing with the measurement definitions can be found in the file `Metric_Types.h`. The definitions generally look like those shown in listing B.10.

Listing B.10: Example measure definitions

```
typedef Object_Metric<import_dir,
                      constraint<import_constraint<C,M,L>,
                                 export_constraint<C,M> > > IC_OD;
typedef Object_Metric<export_dir,
                      constraint<import_constraint<C,M>,
                                 export_constraint<C> > >   EC_OC;
typedef Class_Metric<import_dir,
                     constraint<import_constraint<C,M>,
                                export_constraint<C,M> > >  IC_CM;
```

In the next sections we will give a brief introduction to the meaning of the various template parameters.

Measure Classes

The template definitions in the example might seem daunting. However, they are not as complicated as one might assume.

This example shows two object-level measures (xx_Ox) acting on the set of messages ME, and one class-level measure (x_Cx) which is used to analyse IV.

Whether IV or ME is analysed is determined by the class used in the definition. `Object_Metric` measures ME, while `Class_Metric` does the same for IV.

Both classes are descendants of `Metric`, and they both override the function
`virtual void calculateSingle(Class* c)`, which calculates coupling for a single class.

Direction Template Parameter

The first template parameter represents the direction of the measurement. It can be set to either `import_dir` or `export_dir`, depending on whether we want to measure import or export coupling.

As seen in the example declarations in listing B.10, this parameter does not depend on whether we are measuring object- or class-level coupling.

The direction parameter corresponds to the first letter of the measurement definition name. Ix_xx always means import, while Ex_xx always refers to export coupling.

Constraint Template Parameter

The last parameter controls the uniqueness criteria of the various measurements. It consists of a class `constraint`, which needs two template parameters to inherit from. These two parameters should always be set to the values seen in the example code (`import_constraint` and `export_constraint`).

The exact reasons for this complicated syntax is best left unexplained. It mostly has to do with overcoming limitations in the C++ template syntax. Interested readers can refer to Vandevoorde and Josuttis and their description of function object composition [VJ02, p. 445].

But explaining the arguments to the `import_` and `export_constraint` templates is vital. Especially the meaning of the letters `C`, `M` and `L`. It is, however, easier to understand their significance having compared the C++ code in listing B.10 to the theoretical definitions from chapter 2.

$$IC\_OD(c) = \{(m,c,l,c',m') | (\forall (o,c) \in R_{OC})(\exists (o',c') \in R_{OC}, l \in \mathbb{N}) \\ c \neq c' \wedge (o,m,l,o',m') \in ME\} \tag{B.2}$$

$$EC\_OC(c) = \{(m',c',c) | (\forall (o,c) \in R_{OC})(\exists (o',c') \in R_{OC}, l \in \mathbb{N}) \\ c \neq c' \wedge (o',m',l,o,m) \in ME\} \tag{B.3}$$

$$IC\_CM(c) = \{(m,c,m',c') | (\exists (m,c),(m',c') \in R_{MC}) \\ c \neq c' \wedge (m,c,m',c') \in IV\} \tag{B.4}$$

These equations correspond to the previous C++ definition. The `C`, `M` and `L` parameters from listing B.10 can be seen in the first parenthesis of the formal definitions. Armed with this knowledge it should be relatively easy to understand the relationship between the formal definitions and the source code excerpt.

In listing B.10 the letters (`C,M,L`) represents function objects, which can be combined to form uniqueness criteria for STL sets. Such sets can be populated with the contents of either ME or IV, depending on the class used in the declaration of the measurement.

The numerical value of any coupling measurement is simply the cardinality of the set with uniqueness determined by a combination of the letters.

B.9 Future Work

Most software is created under some form of pressure or time limit. Jdissect is no exception to this rule. There is still a lot of work remaining before the application can possibly be called completed.

In this section we will attempt to address some of what we deem to be the most serious deficiencies.

- **Scalpel** - the interactive testcase control tool has not been thoroughly tested with `libjdissect.so`.

- **Testcases** - Generating measures based on multiple datasets containing more than one testcase does not work. In other words, test-cases are not functional in `mcalc`.

- **Sanity checking event code** - Some of the event code in Jdissect needs sanity checks dealing with event order. Presently tested JVM implementations have no problem with the existing code. But it is possible that future implementations will need stricter checks.

- **Class load events** - Class load events from both the debug and profiling APIs are used. All tested Java versions seem to send the class load profiling events before they send class prepare debugging event. This might change. At present there are no caching mechanisms in place to handle receiving the events out of order.

- **Testing with different Java VM implementations** - As of now Jdissect has only been fully tested using two versions IBM's Java VM. Build 1.3.1, J2RE 1.3.1 IBM build cxia32131-20021102 and build 1.4.0, J2RE 1.4.0 IBM build cxia32140-20020917a. Some rudimentary tests were also done using Sun's Java implementation (build 1.4.2-b28) on Solaris.

- **Threads and locking** - Using thread local storage as temporary storage for messages data before updating the core model might improve performance. Especially in the case of heavily multi-threaded applications.

- **Performance improvements** - Currently, not enough work has gone into optimisation of Jdissect. Such improvements might make it possible to collect data from production systems.

- **Visibility of class variables** - Much of the code dealing with accessing and protecting class attributes should be cleaned up. At present many variables are declared `public`.

- **Use of `typedef`** - Some core model classes do not use the appropriate `typedef`'s for declaring sets of objects. This is due to dependency problems between the `.h` files of the Jdissect project. Given some time it should be possible to fix this.

- **HybridSetMap** - This class performs the functions of both a set and a map simultaneously. It has a large overhead, as it is implemented using both STL `set` and `map`. The problem of an effective data-structure that caters to the need for key-based lookup with unique elements is being investigated by the Boost C++ group. Availability of such a container would decrease the memory requirements of Jdissect.

- **Partial data writes** - Monitoring a Java application over days or months using Jdissect is presently impossible due to memory requirements. This could be solved by writing ME and parts of O to disk while an application was executing.

- **Static methods** - At present static methods are not handled by Jdissect. Implementing this could possibly give rise to no less than 30 new measures. An informal analysis of JBoss (version 3.0.4) showed that roughly 6`static`.$When spread over 30 new measures these 6$ out 24553 methods) will probably not contain much relevant information.

- **Visitor pattern** - Use of the Visitor pattern did not become as widespread as anticipated. A more efficient storage algorithm might be feasible.

- **Automated tests** - Whenever a modification is made to Jdissect we have to verify that the implementation still works according to specification. If the application is to be developed further it would be economical to include a set of automated tests performing the different verification steps laid out in section 3.4.

- **Identity definitions** - At present all core model classes have identities of type `long`. There will never be a need for using negative numbers as identifiers, so all identities should be changed to `unsigned long`. This will allow even larger amounts of data to be collected.

References

[But97] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1994.

[Lia99] S. Liang. *The Java Native Interface - Programmer's Guide and Spesification.* Addison-Wesley, 1999.

[MDS01] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide, 2nd. ed.* Addison Wesley, 2001.

[Mey01] S. Meyer. *Effective STL.* Addison-Wesley, 2001.

[OW99] S. Oaks and H. Wong. *Java Threads, 2nd. ed.* O'Reilly, 1999.

[PSLM01] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library.* Prentice Hall, 2001.

[SL94] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language, 3rd. ed.* Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1997.

[VJ02] D. Vandevoorde and N. M. Josuttis. *C++ Templates - The Complete Guide.* Addison-Wesley, 2002.

Appendix C

Extra material, source code and configuration

C.1 Polymorphism and Coupling

The source code provided below is an example what might confuse static coupling analysis tools. The problem with this code is that it will seem like `UsingAbstract` is coupled twice to `AbstractClass` (the argument of the `using_abstract` function).

As seen from the code, there is no possible way of knowing which class `c1` and `c2` will be set to.

Listing C.1: Abstract coupling Java code

```
import java.lang.Math;

abstract class AbstractClass {
    abstract void method();
}

class ConcreteClass1 extends AbstractClass {
    void method() {
        System.out.println("ConcreteClass 1");
    }
}

class ConcreteClass2 extends AbstractClass {
    void method() {
        System.out.println("ConcreteClass 2");
    }
}

class UsingAbstract {
    void using_abstract(AbstractClass c) {
        c.method();
    }
}

class ConcreteFactory {
    public static AbstractClass getConcreteClass() {
        if( Math.random() < 0.5 )
            return new ConcreteClass1();
        else
            return new ConcreteClass2();
    }
}

public class Exec {
    public static void main(String[] argv) {
        AbstractClass c1 = ConcreteFactory.getConcreteClass();
```

```
        AbstractClass c2 = ConcreteFactory.getConcreteClass();
        UsingAbstract ua = new UsingAbstract();

        ua.using_abstract(c1);
        ua.using_abstract(c2);
    }
}
```

C.2 Taxonomy of Software Metrics

- Software Metrics
  - Requirement Analysis Metrics
    - Project/Requirement/Risk Management
    - Problem Definition Text Analysis
    - Requirement Analysis
  - Specification Metrics
    - Cost/Effort/Size estimation
    - COCOMO
    - Function/Object/Process Points
    - Formal Specification
  - Design Metrics
    - Software Systems/ Architecture
    - Modularization Measurement
    - Software Components Measures
    - Software Agents Measurement
    - Web Measurement
    - Pseudocode Measures
    - Communication/Interaction Measures
    - Object-Oriented Design Measures
    - Review/Inspection/Audits Measures
    - Information Measures
  - Code Metrics
    - Halsteads Software Science
    - McCabes Cyclomatic Number
    - Source Code Measures
    - Formal Analysis and Grammars
    - Control Flow Measures
    - Data (Flow) Measures
    - Hybrid Measures
    - Concurrency Measurement

        * Object-Oriented Programming
        * Functional Programming
        * Logical Programming
        * Visual Programming
    - Test Metrics
        * Test Coverage/ OO Testing
        * Reliability Measurement
        * Security Measurement
        * Performance Measurement
    - Maintenance Metrics
        * Modifiability/Portability Measures
        * Reusability Measures
        * Programmers Productivity Measures
        * Evaluation/Certification Measures

Source: http://ivs.cs.unimagdeburg.de/sweng/us/bibliography/bib_main.shtml
Originally a bibliography list published by Reiner R. Dumke, at the Institute for Distributed Systems, Otto-von-Guericke-University of Magdeburg.

C.3 Downloading Versions from the Velocity CVS Repository

Listing C.2: cvs-snarf.pl

```
#!/usr/bin/perl

# cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login

my @v_tags = ('VEL_1_3_RC1',
              'VEL_1_3_1_RC2',
              'VEL_1_3_1_RC1',
              'VEL_1_3_1',
              'VEL_1_3',
              'VEL_1_2_RC3',
              'VEL_1_2_RC2',
              'VEL_1_2_RC1',
              'VEL_1_2',
              'VEL_1_1_RC2',
              'VEL_1_1_RC1',
              'VEL_1_1',
              'V_1_0B2',
              'V_1_0B1',
              'V_1_0_1_RC1',
              'V_1_0_1',
              'V_1_0');

my $repository = ':pserver:anoncvs@cvs.apache.org:/home/cvspublic';
my $dir;

foreach $version_tag (@v_tags) {
    $dir = $version_tag;
    $dir =~ s/(VEL|V)\_/velocity\-/;
    $dir =~ s/\_/./g;
    $dir =~ s/\.([A-Z]+)(\d+)/$1$2/;
    $dir = lc($dir);

    print "Getting version $version_tag from cvs...\n";
```

```
    `cvs -z3 -d $repository co -r $version_tag jakarta-velocity`;
    `mv jakarta-velocity $dir`;

}
```

C.4 XSLT Stylesheet used to Transform `testcases.xml` for each Velocity Version

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">
<xsl:output method="xml" indent="no"/>

<!-- output configuration needed for the VM to use the library -->
<xsl:template name="jdissectmacro">
  <xsl:param name="version"/>
  <xsl:param name="testName"/>

  <xsl:variable name="path">
    <xsl:value-of select="concat('/tmp/', $version, '/', $testName)"/>
  </xsl:variable>

  <xsl:element name="jvmarg">
    <xsl:attribute name="value">-Xrunjdissect:
    <xsl:value-of select="$path"/></xsl:attribute>
  </xsl:element>
  <xsl:element name="jvmarg">
    <xsl:attribute name="value">-Xdebug</xsl:attribute>
  </xsl:element>
  <xsl:element name="jvmarg">
    <xsl:attribute name="value">-Xnoagent</xsl:attribute>
  </xsl:element>
  <xsl:element name="jvmarg">
    <xsl:attribute name="value">-Djava.compiler=NONE</xsl:attribute>
  </xsl:element>
  <xsl:element name="env">
    <xsl:attribute name="key">LD_LIBRARY_PATH</xsl:attribute>
    <xsl:attribute name="value">${jdissect.path}</xsl:attribute>
  </xsl:element>
</xsl:template>

<!-- the actual document parsing code -->

<!-- properties for metrics. -->
<!-- Should be included at top of testcases.xml file -->
<xsl:template match="property[position() = last()]">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
  <xsl:element name="property">
    <xsl:attribute name="name">jdissect.path</xsl:attribute>
    <xsl:attribute name="value">/home/audunf/hfag/src3</xsl:attribute>
  </xsl:element>
  <xsl:element name="property">
    <xsl:attribute name="name">mcalc.path</xsl:attribute>
    <xsl:attribute name="value">
      /home/audunf/hfag/src3/metric_calc
    </xsl:attribute>
  </xsl:element>
  <xsl:element name="property">
    <xsl:attribute name="name">mcalc.output.path</xsl:attribute>
    <xsl:attribute name="value">
      /home/audunf/velocity-versions/metric_data
    </xsl:attribute>
  </xsl:element>
</xsl:template>

<!-- add extra parameters to each java target -->
<xsl:template match="project/target/java">
  <xsl:copy>
    <xsl:call-template name="jdissectmacro">
      <xsl:with-param name="version">
        ${version}
      </xsl:with-param>
      <xsl:with-param name="testName">
        <xsl:value-of select="../@name"/>
      </xsl:with-param>
    </xsl:call-template>
    <xsl:copy-of select="@*"/>
```

```
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

<!-- copy all default handler -->
 <xsl:template match="*">
   <xsl:copy>
     <xsl:copy-of select="@*"/>
     <xsl:apply-templates/>
   </xsl:copy>
 </xsl:template>

</xsl:stylesheet>
```

C.5 `filter.conf` used to analyse Velocity

```
# Config file for filter
# remember - all regexes are padded with .* at the start.
# One can therefore write "include Vector" instead of "include .*Vector"

# for velocity, exclude all but the core app.
exclude .
exclude_count .

# actual classes right under velocity package
include org/apache/velocity/Template
include org/apache/velocity/VelocityContext
# packages
include org/apache/velocity/app/.*
include org/apache/velocity/app/event/.*
include org/apache/velocity/app/tools/.*
include org/apache/velocity/context/.*
include org/apache/velocity/exception/.*
include org/apache/velocity/runtime/.*
include org/apache/velocity/runtime/configuration/.*
include org/apache/velocity/runtime/directive/.*
include org/apache/velocity/runtime/exception/.*
include org/apache/velocity/runtime/log/.*
include org/apache/velocity/runtime/parser/.*
include org/apache/velocity/runtime/parser/node/.*
include org/apache/velocity/runtime/resource/.*
include org/apache/velocity/runtime/resource/loader/.*
include org/apache/velocity/runtime/visitor/.*
include org/apache/velocity/util/.*
include org/apache/velocity/util/introspection/.*

# actual classes right under velocity package
include_count org/apache/velocity/Template
include_count org/apache/velocity/VelocityContext
# packages
include_count org/apache/velocity/app/.*
include_count org/apache/velocity/app/event/.*
include_count org/apache/velocity/app/tools/.*
include_count org/apache/velocity/context/.*
include_count org/apache/velocity/exception/.*
include_count org/apache/velocity/runtime/.*
include_count org/apache/velocity/runtime/configuration/.*
include_count org/apache/velocity/runtime/directive/.*
include_count org/apache/velocity/runtime/exception/.*
include_count org/apache/velocity/runtime/log/.*
include_count org/apache/velocity/runtime/parser/.*
include_count org/apache/velocity/runtime/parser/node/.*
include_count org/apache/velocity/runtime/resource/.*
include_count org/apache/velocity/runtime/resource/loader/.*
include_count org/apache/velocity/runtime/visitor/.*
include_count org/apache/velocity/util/.*
include_count org/apache/velocity/util/introspection/.*

# are excluded (don't want to measure coupling for the testcases)
# org/apache/velocity/test
# org/apache/velocity/test/misc
# include org/apache/velocity/test/provider/.*
# include_count org/apache/velocity/test/provider/.*

# tasks that are not in the build package,
# but are still tests (the 'oracle' based tests)

# to exclude:
```

```
# org.apache.velocity.anakia.AnakiaTask
include org/apache/velocity/anakia/Escape
include org/apache/velocity/anakia/Tree
include org/apache/velocity/anakia/NodeList.*
include org/apache/velocity/anakia/TreeWalker
include org/apache/velocity/anakia/OutputWrapper
include org/apache/velocity/anakia/AnakiaJDOMFactory
include org/apache/velocity/anakia/XPathTool
include org/apache/velocity/anakia/AnakiaElement
# include org/apache/velocity/anakia/AnakiaTask
include org/apache/velocity/anakia/XPathCache
include_count org/apache/velocity/anakia/Escape
include_count org/apache/velocity/anakia/Tree
include_count org/apache/velocity/anakia/NodeList.*
include_count org/apache/velocity/anakia/TreeWalker
include_count org/apache/velocity/anakia/OutputWrapper
include_count org/apache/velocity/anakia/AnakiaJDOMFactory
include_count org/apache/velocity/anakia/XPathTool
include_count org/apache/velocity/anakia/AnakiaElement
# include_count org/apache/velocity/anakia/AnakiaTask
include_count org/apache/velocity/anakia/XPathCache

# to exclude:
# org.apache.velocity.texen.ant.TexenTask
include org/apache/velocity/texen/Generator
include_count org/apache/velocity/texen/Generator

include org/apache/velocity/texen/util/.*
include_count org/apache/velocity/texen/util/.*
```

C.6 JVMPI events used by `libjdissect.so`

- `JVMPI_EVENT_CLASS_LOAD` is sent whenenver a new class is loaded by the VM.
- `JVMPI_EVENT_CLASS_UNLOAD` is triggered whenever a class is unloaded. It never seems to occur in practice.
- `JVMPI_EVENT_GC_START` is sent when GC is about to start. The system goes into thread suspended mode. All memory allocation operations should be suspended after handling this event. JvmpiController allocates large buffer in which to place objects that are deallocated or moved.
- `JVMPI_EVENT_GC_FINISH` is sent when GC has ended. System goes into multithreaded mode again. Data cached by JvmpiController is merged into the core model.
- `JVMPI_EVENT_JVM_INIT_DONE` is issued by the VM when its initialization is done. Triggers attempt to set up socket for external control.
- `JVMPI_EVENT_JVM_SHUT_DOWN` is sent when the program exits. Triggers storage of the core model to disk.
- `JVMPI_EVENT_METHOD_ENTRY2` is sent when a method is entered.
- `JVMPI_EVENT_METHOD_EXIT` is triggered whenever a method has completed execution.
- `JVMPI_EVENT_OBJECT_ALLOC` is sent when an object is allocated.

- `JVMPI_EVENT_OBJECT_FREE` is only possible after garbage collection has started (thread suspended mode). It is sent when an object is freed.
- `JVMPI_EVENT_OBJECT_MOVE` is only to happen after garbage collection has started (thread suspended mode). It is sent when an object is moved in the heap.
- `JVMPI_EVENT_THREAD_START` Issued whenever a new thread starts.
- `JVMPI_EVENT_THREAD_END` is triggered whenever a thread is finished executing.

C.7 Intermediate storage file format

Table C.1: File formats use by Jdissect.

File name	**Contains**	**Comment**
attribute	aid of attribute, type, name	*Storage*
class	cid, type	*Storage*
class_ancestor	cid, ancestor cid	*Relationship*
class_attribute	cid of class, aid of attribute	*Relationship*
class_method	cid of class, mid of method	*Relationship*
class_object	cid of class, oid of object	*Relationship*
inoutparam	pid of parameter, type, name	*Storage*
inputparam	pid of parameter, type, name	*Storage*
message	mid of message, isReflexive, location, threadName, exportMethodId, importMethodId, exportObjectId, importObjectId	*Storage*. ExportMethodId, importMethodId and exportObjectId are -1 if method/object was defined as static. ImportObjectId is -1 when if object is static, and -2 if importer was the Java VM.
method	mid of method, signature	*Storage*
method_parameter	mid of method, pid of parameter	*Relationship*
object	oid of object	*Storage*
outputparam	pid of parameter, type, name (always returnValue)	*Storage*
staticattribute	aid of attribute, type, name	*Storage*
testcase	tid of testcase, name	*Storage*
testcase_message	tid of testcase, mid of message	*Relationship*

Bibliography

[ABF03] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. Technical report, Simula Research Laboratory, TR 2003-5/Carleton University, Canada, TR SCE-03-18, 2003.

[ABFar] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, To Appear.

[Adr93] W. R. Adrion. Research methodology in software engineering, summary of the Dagstühl workshop on future directions in software engineering. *ACM SIGSOFT Software Engineering Notes*, 18(1):35–48, January 1993.

[Alb79] A. J. Albrecht. Measuring application developement. pages 83–92, Monterey CA, 1979. Proceedings of IBM Application Developement joint SHARE/GUIDE Symposium.

[AN02] J. Arlow and I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design.* Pearson Education Limited, Edinburgh Gate, Harlow CM20 2JE, 2002.

[And03] B. Anda. *Empirical Studies of Contruction and Application of Use Case Models.* PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2003.

[Ari01] E. Arisholm. *Empirical Assessment of Changability in Object-Oriented Software.* PhD thesis, University of Oslo, Oslo, 2001.

[Ari02] E. Arisholm. Dynamic coupling measures for object-oriented software. In *proc. 8th IEEE Symposium on Software Metrics (METRICS'02)*, pages 33–42. IEEE Computer Society, 4-7 June 2002.

[AS00] E. Arisholm and D. I. K. Sjøberg. Towards a framework for empirical assessment of changeability decay. *The Journal of Systems and Software*, 53(1):3–14, 2000.

[AS03] E. Arisholm and D. I. K. Sjøberg. A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Technical Report TR 2003-6, Simula Research Laboratory, 6 2003.

[ASF04a] The Apache Software Foundation. The Apache Ant project. http://ant.apache.org/, 2004.

[ASF04b] The Apache Software Foundation. The Apache Jakarta project. http://jakarta.apache.org/, 2004.

[ASF04c] The Apache Software Foundation. The Apache Velocity project. http://jakarta.apache.org/velocity/, 2004.

[ASJ01] E. Arisholm, D. I. K. Sjøberg, and M. Jørgensen. Assessing the changeability of two object-oriented design alternatives - a controlled experiment. *Empirical Software Engineering*, 6:231–277, 2001.

[BAJ01] L. Bratthall, E. Arisholm, and M. Jørgensen. Program understanding behaviour during estimation of enhancement effort on small Java programs. In *proc. PROFES 2001 (3rd International Conference on Product Focused Software Process Improvement)*, 2001.

[Bas96] V. R. Basili. The role of experimentation in software engineering: Past, current and future. volume 18 of *Proceedings of 18th International Conference on Software Engineering*, pages 442–449. IEEE, March 25-29 1996.

[BB94] C. L. Brooks and C. G. Buell. A tool for automatically gathering object-oriented metrics. volume 2 of *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference*, pages 835–838. IEEE, NAECON, 23-27 May 1994.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. Proceedings of the Second International Conference on Software Engineering, pages pp. 592–605. IEEE, 1976.

[BDM+95] A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Technical Report EFoCS-17-95, ISERN-96-10, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1995.

[BDM97] L. C. Briand, P. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In *proc. 19th International Conference on Software Engineering (ICSE'97)*, pages 412–421, 1997.

[BDW98] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[BDW99] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan./Feb. 1999.

[BeA95] F. Brito e Abreu. The MOOD metrics set. In *proc. ECOOP'95 Workshop on Metrics*, 1995.

[BEEM96] L. C. Briand, K. El-Emam, and S. Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering: An International Journal*, 1(1):61–88, 1996.

[BJ95] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison Wesley Longman, Reading, Massachusetts, U.S.A., 1995.

[BJY01] J. M. Bieman, D. Jain, and H. J. Yang. OO design patterns, design structure, and program changes: An industrial case study. In *ICSM*, pages 580–, 2001.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Users Guide*. Addison-Wesley, 1998.

[BS98] A. B. Binkley and S. R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *proc. 20th International Conference on Software Engineering (ICSE'98)*, pages 452–455, 1998.

[But97] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[BW02a] L. C. Briand and J. K. Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 59:97–166, 2002.

[BW02b] L. C. Briand and J. K. Wüst. The impact of design properties on developement cost in object-oriented systems. Technical Report TR-99-16, ISERN, 2002.

[BWIL99] L. C. Briand, J. K. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality in object-oriented designs: an industrial case study. In *proc. 21st International Conference of Software Engineering (ICSE'99)*, pages 345–354, 1999.

[BWL99] L. C. Briand, J. K. Wüst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *proc. International Conference on Software Maintenance (ICSM'99)*, pages 475–482, 1999.

[BWW02] L. C. Briand, M. L. Walcelio, and J. K. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, July 2002.

[Cas02] F. Castel. Theory, theory on the wall. *Communications of the ACM*, 45(12):25–26, December 2002.

[CCA86] D. N. Card, V. E. Church, and W. W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, 12(2):264–271, 1986.

[CDK98] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–637, 1998.

[CG93] R. E. Courtney and D. A. Gustafson. Shotgun correlations in software measure. *Software Engineering Journal*, pages 5–13, Jan. 1993.

[CHT02] J. Cahill, J. M. Hogan, and R. Thomas. The Java metrics reporter - an extensible tool for OO software analysis. Ninth Asia-Pacific Software Engineering Conference, pages 507–516. IEEE, 4-6 Dec. 2002.

[CK91] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming: Systems, Languages and Applications*, volume 26, pages 197–211. SIGPLAN Notices, Oct. 1991.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CKK+00] M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. In *proc. Fourth Euromicro Working Conference on Software Maintenance and Reengineering*, pages 45–54, 2000.

[Con01] L. L. Constantine. *The Peopleware Papers, notes on the human side of software*. Prentice Hall Inc., Upper Saddle River, New Jersey 07458, U.S.A., 2001.

[CPM85] D. N. Card, G. T. Page, and F. E. McGarry. Criteria for software modularization. IEEE Eighth International Conference on Software Engineering, pages 372–377. IEEE, 1985.

[CS00] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Systems*, 26(8):786–796, 2000.

[CY91] P. Coad and E. Yourdon. *Object Oriented Design*. Prentice Hall, 1st edition, 1991.

[DBM+96] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[DSWR02] I. S. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis. A review of experimental investigations into object-oriented technology. *Empirical Software Engineering*, 7(3):193–232, 2002.

[Dun98] G. Dunteman. *Principal Component Analysis*. SAGE publications, 1998.

[EE01] K. El-Emam. A methodology for validating software product metrics, 2001.

[EEBGR01] 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.

[Fen94] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[FN99] N. Fenton and M. Neil. Software metrics and risk. FESMA 99 - 2nd European Software Measurement Conference, October 1999.

[FPG94] N. Fenton, S. L. Pfleeger, and R. L. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(4):88–95, 1994.

[FW98] R. J. Freund and W. J. Wilson. *Regression Analysis: statistical modelling of a response variable*. Academic Press, 1998.

[GB04] E. Gamma and K. Beck. Junit - regression testing framework. http://www.junit.org/index.htm, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1994.

[Gla94] R. L. Glass. The software-research crisis. *IEEE Software*, 11(6):42–47, Nov. 1994.

[Goo93] P. Goodman. *Practical Implementation of Software Metrics*. McGraw Hill, London, 1993.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, June 1977.

[HCN98] R. Harrison, S. J. Counsell, and R. V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[HDFW03] T. Husted, C. Dumoulin, G. Franciscus, and D. Winterfeldt. *Struts in Action*. Manning, 2003.

[Her99] J. D. Herbsleb. Metaphorical representation in collaborative software engineering. Proceedings of the international joint conference on Work activities coordination and collaboration, pages 117–126. ACM, February 1999.

[HHL90] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the maintainability of object-oriented software. In *proc. IEEE Region 10 Conference on Computer and Communication Systems (TENCON'90)*, pages 404–409, 1990.

[HM95] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object oriented systems. In *proc. Int. Symp. Applied Corporate Computing*, 1995.

[HM96] M. Hitz and B. Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.

[Hum89] W. S. Humphrey. *Managing the Software Process*. Addison-Wesley, Reading Mass., 1989.

[IC01] JTC 1-SC 7 ISO Commitee. ISO/IEC 9126-1:2001 - software engineering - product quality - part 1: Quality model. Technical report, ISO/IEC, 2001.

[IC03a] JTC 1/SC 7 ISO Commitee. ISO/IEC 9126-2:2003 - software engineering - product quality - part 2: External metrics. Technical report, ISO/IEC, 2003.

[IC03b] JTC 1/SC 7 ISO Commitee. ISO/IEC 9126-3:2003 - software engineering - product quality - part 3: Internal metrics. Technical report, ISO/IEC, 2003.

[Jar01] P. Jarvinen. *On Research Methods*. Tiedekirjakauppa TAJU, 2001.

[JS02] D. R. Jeffery and L. Scott. Has twenty-five years of empirical software engineering made a difference? In P. Strooper and P. Muenchaisri, editors, *Proceedings of the Asia-Pacific Software Engineering Conference, Gold Coast Australia*, pages 539–546. IEEE Computer Society, Dec. 2002.

[Kit90] B. A. Kitchenham. *Measuring Software Development*. Software Reliability Handbook. Elsevier Press, 1990.

[Kit96a] B. A. Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, January 1996.

[Kit96b] B. A. Kitchenham. *Software Metrics: Measurement for Software Process Improvement*. Blackwell, 1996.

[Knu97] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley Longman, 3rd. edition, 1997.

[KPF95] B. A. Kitchenham, S. L. Pfeelger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[KPP95] B. A. Kitchenham, L. Pickard, and S.L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El-Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[LH93a] W. Li and S. Henry. Maintenance Metrics for The Object Oriented Paradigm. In *Proc. 1st IEEE Int. Software Metrics Symposium*, pages 52–60, Baltimore, Md, 1993.

[LH93b] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 1993.

[Lia99] S. Liang. *The Java Native Interface - Programmer's Guide and Spesification*. Addison-Wesley, 1999.

[LLWW95] Y. S Lee, B. S. Liang, S.F. Wu, and F. J. Wang. Measuring coupling and cohesion of object-oriented programs based on information flow. In *proc. International Conference on Software Quality*, 1995.

[MA85] Lehman M. M. and Belady L. A. Program evolution: Processes of software change. 1985.

[MBB+97] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, and M. V. Zelkowitz. Comments on "Towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):187–188, 1997.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[MDS01] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide, 2nd. ed.* Addison Wesley, 2001.

[Mey01] S. Meyer. *Effective STL*. Addison-Wesley, 2001.

[MH96] J. C. Munson and G. A. Hall. Estimating test effectiveness with dynamic complexity measurement. *Empirical Software Engineering*, (1):279–305, 1996.

[Mil] J. Miller. Replicating software engineering experiments: A poisoned chalice or the holy grail. Draft.

[NR68] P. Naur and B. Randell, editors. *Software Engineering - Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO.

[OW99] S. Oaks and H. Wong. *Java Threads, 2nd. ed.* O'Reilly, 1999.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1052–1058, December 1972.

[PSLM01] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library.* Prentice Hall, 2001.

[PWC95] M. C. Paulk, C. V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Carnegie Mellon University / Software Engineering Institute / Addison-Wesley, Reading Mass., 1995.

[SAA+02] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokác. Conducting realistic experiments in software engineering. Proceedings of the 2002 International Symposium on Empirical Software Engineering. IEEE, 2002.

[SDPK01] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *TOOLS Europe 2001, Zurich, Switzerland*, March 2001.

[SG94] The Standish Group. The chaos report. Technical report, The Standish Group, 1994.

[SKM01] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software Practice & Experience*, (31):371–394, Feb. 2001.

[SL94] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard, 1994.

[SMC74] W. Stevens, G. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Som98] I. Sommerville. *Software Engineering.* Addison-Wesley, 1998.

[Ste93] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1993.

[Str97] B. Stroustrup. *The C++ Programming Language, 3rd. ed.* Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1997.

[SYM00] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000), Zurich, Switzerland*, March 2000.

[TZ92] J. Tian and M. V. Zelkowitz. A formal program complexity model and its application. *Journal of Systems Software*, 17:253–266, 1992.

[VJ02] D. Vandevoorde and N. M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2002.

[Wey88] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.

[WK00] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in C++ application software. *J. Syst. Softw.*, 52(2-3):157–164, 2000.

[YAR99] S. M. Yacoub, H. H. Ammar, and T. Robinson. Dynamic metrics for object-oriented designs. pages 60–61, 1999.

[YAR00] S. M. Yacoub, H. H. Ammar, and T. Robinson. A methodology for architectural-level risk assessment using dynamic metrics. In *proc. 11th International Symposium on Software Reliability Engineering*, pages 210–221, 2000.

[ZW98] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *Computing Practices*, pages 23–31, May 1998.