Compendium of Software Quality Standards and Metrics – Version 1.0

Rüdiger Lincke – Rudiger.Lincke@arisa.se
Welf Löwe – Welf.Lowe@arisa.se

April 3, 2007

Chapter 1

Introduction

Goal of the *Compendium of Software Quality Standards and Metrics* is to provide an information resource connecting software quality standards with well-known software metrics. Currently, the compendium describes

- 37 software quality attributes (factors, criteria), and
- 23 software quality metrics.

Please refer to the "How to use the Compendium", Section 1.1, to learn how to efficiently consult the compendium.

The `quality-metrics-compendium.pdf` is a printer-friendly version of the compendium. The `quality-metrics-matrix.pdf` gives an overview of the relations between quality properties and metrics.

This compendium is meant to be a "living document". Feel free to contribute with new metrics and comments, corrections, and add-ons to already defined ones. References to validating experiments or industrial experiences are especially welcome. Please refer to the *"How to use the Compendium"*, Section 1.1, to learn how to submit contributions.

1.1 How to use the Compendium

1.1.1 I want to consult the Compendium

The compendium has two main indices: *"Software Quality ISO Standards"*, (cf. Section 2) and *"Software Quality Metrics"* (cf. Section 3). The first index lists quality attributes and sub-attributes. Each attribute (factor)

and sub-attribute (criterion) is first defined and then related to a number of metrics allowing its assessment. The second index lists quality metrics. Each metric is defined and then related to a number of quality attributes (factors) and sub-attributes (criteria) that it supports to assess.

1.1.2 I want to contribute to the Compendium

All contributions will be edited before publication. Please submit contributions to Rüdiger Lincke (`rudiger.lincke@arisa.se`).

If you want to describe a new metric, please use the description schema for metrics, cf. Section 3. If you want to comment, correct, or add to an already defined metric, please refer to the metric by its handle and indicate the items in the description schema for metrics, cf. Section 3, your contribution applies to. All other kinds of contributions are welcome as well.

1.2 History of the Compendium

2007-03 Rüdiger Lincke prepares the current version of the compendium for publication.

2007-03 Rüdiger Lincke edits and updates several sections of the compendium related to the Software Quality Model. Connections between metrics and criteria are updated.

2007-02 Rüdiger Lincke edits and updates several sections of the compendium related to metrics. The layout of metric sections is improved. Connections between new metrics and criteria in SQM are updated.

2006-04 through 2006-12 Rüdiger Lincke adapts existing metric definitions to new description schema, based on CMM10, adding metrics Ca 3.2.2, Ce 3.2.2, I 3.2.2, CF 3.2.2.

2005-09-27 through 2005-11-22 Rüdiger Lincke adds Number Of Children (3.2.1), Coupling Between Objects (3.2.2), Data Abstraction Coupling (3.2.2), Package Data Abstraction Coupling (3.2.2), Message Passing Coupling (3.2.2), Number Of local Methods (3.1.2), Change Dependency Between Classes (3.2.2), Locality of Data (3.2.2), Lack of COhesion in Methods (3.2.3), Improvement of LCOM (3.2.3), Tight Class Cohesion (3.2.3).

2005-09-26 Rüdiger Lincke adds Response For a Class 3.1.3

2005-08-18 Welf Löwe adds sub-classes for metrics and Lack of Documentation 3.3.1 metric.

2005-08-12 Welf Löwe adds Lines of Code 3.1.1, Number of Attributes and Methods 3.1.2, Depth of Inheritance Tree 3.2.1 metrics; also adds an alternative definition of McCabe Cyclomatic Complexity 3.1.3.

2005-08-03 Welf Löwe changes the initial classification of metrics. The structure of the compendium is adapted accordingly. The compendium moves to Latex and HTML and is published online.

2005-08-01 Rüdiger Lincke produces the first version of the compendium as Word document containing definitions of the Software Quality ISO Standards, cf. Section 2, McCabe Cyclomatic Complexity 3.1.3 and Weighted Method Count 3.1.3 as initial metrics.

2005-06-16 Rüdiger Lincke and Welf Löwe sketch goals and structure of the compendium.

Chapter 2

Software Quality ISO Standards

The ISO/IEC 9126 standard describes a software quality model which categorizes software quality into six characteristics (factors) which are sub-divided into sub-characteristics (criteria). The characteristics are manifested externally when the software is used as a consequence of internal software attributes.

The internal software attributes are measured by means of internal metrics (e.g., monitoring of software development before delivery). Examples of internal metrics are given in ISO 9126-3. The quality characteristics are measured externally by means of external metrics (e.g., evaluation of software products to be delivered). Examples of external metrics are given in ISO 9126-2.

Our work focuses on the assessment of the internal quality of a software product as it can be assessed upon the source code. In the following we describe the general quality model as defined in ISO 9126-1 with its properties and sub-properties. Details of the internal aspect of this model are given as well.

2.1 Functionality

The *functionality* characteristic allows to draw conclusions about how well software provides desired functions. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in

question satisfies functional requirements.

2.1.1 Suitability

The *suitability* sub-characteristic allows to draw conclusions about how suitable software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude the presence and appropriateness of a set of functions for specified tasks.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

2.1.2 Accuracy

The *accuracy* sub-characteristic allows to draw conclusions about how well software achieves correct or agreeable results. It correlates with metrics which measure attributes of software that allow to conclude about its provision of correct or agreeable results.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

2.1.3 Interoperability

The *interoperability* sub-characteristic allows to draw conclusions about how well software interacts with designated systems. It correlates with metrics which measure attributes of software that allow to conclude about its ability to interact with specified systems.

Highly Related Metrics

- none related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,

- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- Ca, Afferent Coupling 3.2.2,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.

2.1.4 Security

The *security* sub-characteristic allows to draw conclusions about how secure software is. It correlates with metrics which measure attributes of software that allow to conclude about its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- Ca, Afferent Coupling 3.2.2,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.

2.1.5 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

2.2 Reliability

The *reliability* characteristic allows to draw conclusions about how well software maintains the level of system performance when used under specified conditions. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies reliability requirements.

2.2.1 Maturity

The *maturity* sub-characteristic allows to draw conclusions about how mature software is. It correlates with metrics which measure attributes of software that allow to conclude about the frequency of failure by faults in the software.

Highly Related Metrics

- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- Lack Of Documentation (LOD) 3.3.1,

2.2.2 Fault-tolerance

The *fault-tolerance* sub-characteristic allows to draw conclusions about how fault-tolerant software is. It correlates with metrics which measure attributes of software that allow to conclude on its ability to maintain a specified level of performance in case of software faults or infringement of its specified interface.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- Ca, Afferent Coupling 3.2.2,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,

- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.

2.2.3 Recoverability

The *recoverability* sub-characteristic allows to draw conclusions about how well software recovers from software faults or infringement of its specified interface. It correlates with metrics which measure attributes of software that allow to conclude on its ability to re-establish its level of performance and recover the data directly affected in case of a failure.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,

- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.

2.2.4 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

2.3 Usability

The *usability* characteristic allows to draw conclusions about how well software can be understood, learned, used and liked by the developer. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies usability requirements.

Remarks
See also Usability for Reuse 2.4.

2.3.1 Understandability

The *understandability* sub-characteristic allows to draw conclusions about how well users can recognize the logical concepts and applicability of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for recognizing the logical concepts and applicability. Users should be able to select a software product which is suitable for their intended use.

Remarks
See also Understandability for Reuse 2.4.1.

2.3.2 Learnability

The *learnability* sub-characteristic allows to draw conclusions about how well users can learn the application of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for learning the application of software.

Remarks
See also Learnability for Reuse 2.4.2.

2.3.3 Operability

The *operability* sub-characteristic allows to draw conclusions about how well users can operate software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for operation and operation control.

Remarks
See also Operability for Reuse – Programmability 2.4.3.

2.3.4 Attractiveness

The *attractiveness* sub-characteristic allows to draw conclusions about how attractive software is to the user. It correlates with metrics which measure attributes of software that allow to conclude on the capability of the software product to be attractive to the user.

Remarks
See also Attractiveness for Reuse – Programmability 2.4.4.

2.3.5 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

Remarks
See also Compliance for Reuse – Programmability 2.4.5.

2.4 Re-Usability

Usability 2.3 is defined to be the characteristic that allows to draw conclusions about how well software can be understood, learned, used and liked by the developer.

Remarks
Re-Usability is a special case of the quality factor Usability 2.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as usability of software for integration in other systems, e.g., usability of libraries and components. Here the user is a software engineer/developer.

Hence, internal usability metrics are used for predicting the extend of which the software in question can be understood, learned, operated, is attractive and compliant with usability regulations and guidelines *where here using means integrating it in a larger software system.*

2.4.1 Understandability for Reuse

Understandability 2.3.1 is defined as the attributes of software that bear on the users' efforts for recognizing the logical concept and its applicability.

Remarks
Understandability for Reuse is a special case of the quality criterium Understandability 2.3.1 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as understandability of software for a software engineer/developer with the purpose of integrating it in new systems, e.g., understandability of libraries and components.

Hence, software engineers/developers should be able to select a software product which is suitable for their intended use. Internal understandability reuse metrics assess whether new software engineers/developers can understand: whether the software is suitable; how it can be used for particular tasks.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- either not related, or not evaluated

2.4.2 Learnability for Reuse

Learnability 2.3.2 is defined as the attributes of software that bear on the users' efforts for learning its application.

Remarks
Learnability for Reuse is a special case of the quality criterium Learnability 2.3.2 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as learnability of software for a software engineer or developer with the purpose of integrating it in new systems, e.g., learnability of libraries and components.

Hence, internal learnability metrics assess how long software engineers or developers take to learn how to use particular functions, and the effectiveness of documentation. Learnability is strongly related to understandability, and understandability measurements can be indicators of the learnability potential of the software.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- LOC, Lines of Code 3.1.1,
- RFC, Response For a Class 3.1.3,
- Ca, Afferent Coupling 3.2.2,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

2.4.3 Operability for Reuse – Programmability

Operability 2.3.3 is defined as the attributes of software that bear on the users' efforts for operation and operation control.

Remarks
Operability for Reuse is a special case of the quality criterium Operability 2.3.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as programmability of software, i.e., the development effort

for a software engineer/developer to integrate it in new systems, e.g., programmability software using libraries and components.

Hence, internal programmability metrics assess whether software engineers/developers can integrate and control the software.

General operability metrics can be categorized by the dialog principles in ISO 9241-10: suitability of the software for the task; self-descriptiveness of the software; controllability of the software; conformity of the software with user expectations; error tolerance of the software; suitability of the software for individualization. This also applies to operability in a reuse context.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- LOC, Lines of Code 3.1.1,
- RFC, Response For a Class 3.1.3,
- Ca, Afferent Coupling 3.2.2,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,

- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

2.4.4 Attractiveness for Reuse

Attractiveness 2.3.4 is defined as the attributes of software that bear on the capability of the software product to be attractive to the user.

Remarks
Attractiveness for Reuse is a special case of the quality criterium Attractiveness 2.3.4 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as attractiveness of software for a software engineer /developer to integrating it in new systems, e.g., attractiveness of libraries and components.

Internal attractiveness metrics assess the appearance of the software. In a reuse context, they will be influenced by factors such as code and documentation layout. In addition to appearance, internal attractiveness metrics for reuse also assess size and complexity of the reused code.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,

- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2.
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

Related Metrics

- Lack Of Documentation (LOD) 3.3.1.

2.4.5 Compliance for Reuse

Compliance 2.3.5 is defined as the attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions.

Remarks
Compliance for Reuse is a special case of the quality criterium Compliance 2.3.5 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as compliance to re-use standards or conventions or regulations in laws and similar prescriptions.

Internal compliance metrics assess adherence to standards, conventions, style guides or regulations relating to re-usability.

2.5 Efficiency

The *efficiency* characteristic allows to draw conclusions about how well software provides required performance relative to amount of resources used. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies efficiency requirements.

2.5.1 Time behavior

The *time behavior* sub-characteristic allows to draw conclusions about how well the time behavior of software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude about the time behavior of the software (or parts of it) in combination with the computer system during testing or operating.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,

- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

2.5.2 Resource utilization

The *resource utilization* sub-characteristic allows to draw conclusions about the amount of resources utilized by the software. It correlates with metrics which measure attributes of software that allow to conclude about the amount and duration of resources used while performing its function.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

2.5.3 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

2.6 Maintainability

The *maintainability* characteristic allows to draw conclusions about how well software can be maintained. It can be used for assessing, controlling and predicting the effort needed to modify the software product (or parts of it) in question.

2.6.1 Analyzability

The *analyzability* sub-characteristic allows to draw conclusions about how well software can be analyzed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,

- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- NOC, Number Of Children 3.2.1.

2.6.2 Changeability

The *changeability* sub-characteristic allows to draw conclusions about how well software can be changed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for modification, fault removal or for environmental change.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- either not related, or not evaluated

2.6.3 Stability

The *stability* sub-characteristic allows to draw conclusions about how stable software is. It correlates with metrics which measure attributes of software that allow to conclude about the risk of unexpected effects as result of modifications.

Highly Related Metrics

- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- Ca, Afferent Coupling 3.2.2,
- Lack Of Documentation (LOD) 3.3.1.

2.6.4 Testability

The *testability* sub-characteristic allows to draw conclusions about how well software can be tested and is tested. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for validating the software and about the test coverage.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,

- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- NOC, Number Of Children 3.2.1.

2.6.5 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

2.7 Portability

The *portability* characteristic allows to draw conclusions about how well software can be ported from one environment to another. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies portability requirements.

2.7.1 Adaptability

The *adaptability* sub-characteristic allows to draw conclusions about how well software can be adapted to environmental change. It correlates with metrics which measure attributes of software that allow to conclude about the amount of changes needed for the adaptation of software to different specified environments.

Highly Related Metrics

- LOC, Lines of Code 3.1.1,
- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- RFC, Response For a Class 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- CBO, Coupling Between Objects (CBO) 3.2.2,
- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,

- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- NOC, Number Of Children 3.2.1.

2.7.2 Installability

The *installability* sub-characteristic allows to draw conclusions about how well software can be installed in a designated environment. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed to install the software in a specified environment.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

2.7.3 Co-existence

The *co-existence* sub-characteristic allows to draw conclusions about how well software can co-exist with other software products in the same operational environment. It correlates with metrics which measure attributes of software that allow to conclude about the dependencies, concurrency behavior, or side effects.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

2.7.4 Replaceablity

The *replaceability* sub-characteristic allows to draw conclusions about how well software can replace other software or parts of it. It correlates with metrics which measure attributes of software that allow to conclude about opportunity and effort using it instead of specified other software in the environment of that software.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 3.1.2,
- NOM, Number of local Methods 3.1.2,
- CC, McCabe Cyclomatic Complexity 3.1.3,
- WMC, Weighted Method Count 3.1.3,
- DIT, Depth of Inheritance Tree 3.2.1,
- NOC, Number Of Children 3.2.1,
- Ca, Afferent Coupling 3.2.2,
- Lack Of Documentation (LOD) 3.3.1.

Related Metrics

- LOC, Lines of Code 3.1.1,
- RFC, Response For a Class 3.1.3,
- CBO, Coupling Between Objects (CBO) 3.2.2,

- Change Dependency Between Classes (CDBC) 3.2.2,
- Change Dependency Of Classes (CDOC) 3.2.2,
- Efferent Coupling (Ce) 3.2.2,
- Coupling Factor (CF) 3.2.2,
- Data Abstraction Coupling (DAC) 3.2.2,
- Instability (I) 3.2.2,
- Locality of Data (LD) 3.2.2,
- Message Passing Coupling (MPC) 3.2.2,
- Package Data Abstraction Coupling (PDAC) 3.2.2,
- Lack of Cohesion in Methods (LCOM) 3.2.3,
- Improvement of LCOM (ILCOM) 3.2.3,
- Tight Class Cohesion (TCC) 3.2.3.

2.7.5 Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

Chapter 3

Software Quality Metrics

This part of the document defines standard metrics reported in literature. Moreover, it discusses the applicability of each metric in one of the ISO 9126-3:2003 (factors/criteria) properties/sub properties, Section 2.

Metrics are classified as supporting mainly statements on software *Complexity* (cf. Section 3.1), software *Architecture and Structure* (cf. Section 3.2), and software *Design and Coding* (cf. Section 3.3).

We use the following description schema for each metric. It defines:

Handle A short name or symbol uniquely identifying the metric. This is the way it is referenced in this document and literature. If there are more than one variation of a metric, additional symbols will be added to distinguish the different variants.

Description A textual description of the metric and what it measures. This description is a summary of how the metric is described in literature. Since it is described in natural language, it is not suitable for an unambiguous definition of the metric.

Scope Identifies the program element the metric applies to. This could be *system* for the whole software system, *package/directory* for entities of a package or a directory, *class/module/file* for classes, modules and files, respectively, and *procedure/function/method* for procedures, functions and methods, respectively. Only the core scope is named.

Definition A formal definition of the metric using set theory and mathematics. The definition is based on the meta-model specified in the compendium. Usually a view is defined as well, see below.

View Definition of the *view*, describing its relevant elements, semantic relations, and the tree grammar.

Scale Defines the scale type of the metric. It is one of *Absolute*, *Rational*, *Interval*, *Ordinal*, and *Nominal*.

Domain Defines the domain of metric values.

Highly Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are assessable with the metric.

Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are somewhat assessable with the metric.

References Gives references to literature, projects, repositories, and programs using, implementing, or describing the metric. Moreover, it points to validating and evaluating experiments investigating its appropriateness.

Since Since which version of the compendium is the metric contained.

The definition of the metrics is based on the common meta-model as defined in Section **??**. For each metric, we define a *view* further abstracting from the common meta-model to provide exactly the information required by that metric. The view is used for the actual metric definition. This approach makes the metric definitions independent from changes of the common meta-model (this is discussed in more detail in Chapter **??**). Formally, views of a metric analysis A are defined as pairs $V^A = \left(G^A, R^A\right)$ and are bound to the common model with a mapping specification α^A. Again, G^A is a tree grammar specifying the set of view entities and their structural containment required by A. R^A is a set of semantic relations over view entities required by A.

The construction of concrete view models follows the same principles as the abstractions from front-end specific to common models: we ignore some common meta-model entity types, which leads to a filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types, and attach remaining relations defined over

filtered nodes to the relevant ancestors of those nodes, as described in detail in [20].

To simplify metric definitions, we define some utility operation(s):

$succ(e, r)$ $[succ^*(e, r), succ^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] successors of a node or set of nodes e over edges (relations) of type r.

$|S|$ – denotes the number of elements in a set S.

$pred(e, r)$ $[pred^*(e, r), pred^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] predecessors of a node or set of nodes e over edges (relations) of type r.

$dist(e, e', r)$ – denotes the distance or set of distances (number of relations) between a node e and a node or set of nodes e'. It is 0, if it e equals e', and ∞, if there is no path between e and e'.

$max(S)$ – denotes the maximum in a set S of values.

$min(S)$ – denotes the minimum in a set S of values.

$scc(e, r)$ – denotes the set of strongly connected components in a set e of elements over edges (relations) of type r.

$outdegree(e, r)$ $[outdegree^+(e, r)]$ – denotes the number of outgoing relations of type r from a node e including [excluding] relations to e itself.

For additional simplification, we define some frequently used sets:

$M(c) = pred(c, \text{methodof})$ – denotes the set of all methods locally declared in a class c, ignoring visibility and other attributes.

$F(c) = pred(c, \text{fieldof})$ – denotes the set of all fields locally declared in a class c, ignoring visibility and other attributes.

3.1 Complexity

Complexity metrics refer to the static, i.e., structural size and complexity of software. They only indirectly measure execution complexity (if at all).

3.1.1 Size

Lines of Code (*LOC*)

Works only with instances of the common meta-model which were created by a front-end calculating this metric and attaching its value as property to the specific elements. Currently only the Eclipse Java Front-end supports this, the UML Front-end can of course not calculate this metric, since there is no source code in UML.

Handle *LOC*

Description Lines of code simply counts the lines of source code (line break characters) of a certain software entity. It is a simple yet powerful metric to assess the complexity of software entities. Since it is depending on code conventions and format, it is critical to use it in generated codes since it may lack of line breaks. Additionally it can only be measured in the source code itself from the front-end and is therefore a front-end side metric.

Scope Method, Class, Compilation unit

View $V^{LOC} = (G^{LOC}, R^{LOC})$

- Grammar $G^{LOC} = (\{\text{scope}^{LOC}\}, \emptyset, \text{scope}^{LOC})$
- Relations $R^{LOC} : \emptyset$
- Mapping α^{LOC}:

$$\begin{aligned} \alpha^{LOC}(\text{Class}) &\mapsto \text{scope}^{LOC} \\ \alpha^{LOC}(\text{Method}) &\mapsto \text{scope}^{LOC} \\ \alpha^{LOC}(\text{CompilationUnit}) &\mapsto \text{scope}^{LOC} \end{aligned}$$

Definition The *LOC* value of a element $s \in scope^{LOC}$ is defined as:

$$LOC(s) = s.loc$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by size.

Understandability for Reuse 2.4.1: Understanding whether a software entity E is suitable for reuse, or not, depends on its size.
Understandability declines with increasing LOC.

Attractiveness 2.4.4: Attractiveness of a software entity E depends on the size of the potentially reused code.
Attractiveness increases with increasing LOC.

Maintainability 2.6 declines with increasing LOC.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in a software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing LOC.

Changeability 2.6.2: Changing a software entity requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing LOC.

Testability 2.6.4: Complete testing requires coverage of all execution paths. The number of possible execution paths of a software entity increases with its size.
Testability declines with increasing LOC.

Portability 2.7 declines with increasing LOC.

Adaptability 2.7.1: As for changeability 2.6.2, the size of a software entity has a direct impact. Each modification requires understanding which is more complicated for large systems.
Adaptability declines with increasing LOC.

Related Software Quality Properties

Functionality 2.1 might increase with increasing LOC.

Interoperability 2.1.3: Interoperability requires to be able to locate the parts of a system responsible for interoperability.

The size of these parts might indicate a better ability to interact.
Interoperability might increase with increasing LOC.

Security 2.1.4: Relating LOC to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing LOC.

Reliability 2.2 might increase with increasing LOC.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs might be left in the software. Therefore, also maturity may be influenced negatively by code size.
Maturity might decline with increasing LOC.

Fault-tolerance 2.2.2: Relating LOC to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing LOC.

Recoverability 2.2.3: Relating LOC to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing LOC.

Re-Usability 2.4 might decrease with increasing LOC.

Learnability for Reuse 2.4.2: Learning if a software entity is suitable for reuse depends on the size and complexity of its interface. LOC is a general size metric, but not specifically assessing interface size and complexity.
Learnability might decline with increasing LOC.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends the complexity of its interface. LOC is a general size metric, but not specifically assessing interface size and complexity.
Programmability might decline with increasing LOC.

Efficiency 2.5 might decline with increasing LOC.

Time Behavior 2.5.1: Static size might indicate a higher execution time due to increased number of instruction cache misses,

long jumps, etc.
Time behavior might get worse with increasing LOC.

Resource Utilization 2.5.2: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing LOC.

Maintainability 2.6 declines with increasing LOC.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size.
Stability might decline with increasing LOC.

Portability 2.7 declines with increasing LOC.

Replaceablity 2.7.4: The substitute of a component must imitate its observable behavior. Large components have the potential of a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component. Interface size and complexity, an attribute directly connected to replaceability, is not specifically assessed by LOC.
Replaceablity might declines with increasing LOC.

References

- LOC is extensively discussed in [13, p22], [2, p22],
- and evaluated in a case study [19],
- it is implemented in the VizzAnalyzer Metrics Suite.

3.1.2 Interface Complexity

Number of Attributes and Methods ($SIZE2$)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $SIZE2$ specific view).

Handle $SIZE2$

Description Number of Attributes and Methods simply counts the number of attributes and methods of a class. It is an object-oriented metric that can be applied to modular languages by considering the number of variables (globally visible in a module) and its number of functions and procedures.

Scope Class

View $V^{SIZE2} = (G^{SIZE2}, R^{SIZE2})$

- Grammar $G^{SIZE2} = (\{\text{class}^{SIZE2}, \text{method}^{SIZE2}, \text{attribute}^{SIZE2}\}, \emptyset, \text{class}^{SIZE2})$
- Relations $R^{SIZE2} : \{\text{attributeof}^{SIZE2} : \text{attribute}^{SIZE2} \times \text{class}^{SIZE2}, \text{methodof}^{SIZE2} : \text{method}^{SIZE2} \times \text{class}^{SIZE2}\}$
- Mapping α^{SIZE2}:

$$\begin{aligned} \alpha^{SIZE2}(\text{Class}) &\mapsto \text{class}^{SIZE2} \\ \alpha^{SIZE2}(\text{IsMethodOf}) &\mapsto \text{methodof}^{SIZE2} \\ \alpha^{SIZE2}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{SIZE2} \\ \alpha^{SIZE2}(\text{Method}) &\mapsto \text{method}^{SIZE2} \\ \alpha^{SIZE2}(\text{Field}) &\mapsto \text{attribute}^{SIZE2} \end{aligned}$$

Definition The $SIZE2$ value of a class $c \in class^{SIZE2}$ is defined as:

$$\begin{aligned} A(c) &= pred(c, \text{attributeof}^{SIZE2}) \\ &\quad \textit{-- set of attributes directly contained in } c \\ M(c) &= pred(c, \text{methodof}^{SIZE2}) \\ &\quad \textit{-- set of methods directly contained in } c \\ SIZE2(c) &= |A(c) \cup M(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by SIZE2.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on its size.
Understandability declines with increasing SIZE2.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the size and complexity of its interface. SIZE2 measures interface size.
Learnability might decline with increasing SIZE2.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends the complexity of its interface. SIZE2 measures interface size.
Programmability might decline with increasing SIZE2.

Attractiveness 2.4.4: Attractiveness of a class depends on the size of the potentially reused code.
Attractiveness increases with increasing SIZE2.

Maintainability 2.6 declines with increasing SIZE2.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing SIZE2.

Changeability 2.6.2: Changing a class requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing SIZE2.

Testability 2.6.4: Complete testing requires coverage of all execution paths. The number of possible execution paths of a system increases with its size.
Testability declines with increasing SIZE2.

Portability 2.7 declines with increasing SIZE2.

Adaptability 2.7.1: As for changeability 2.6.2, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems.
Adaptability declines with increasing SIZE2.

Replaceablity 2.7.4: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by SIZE2.
Replaceablity decline with increasing SIZE2.

Related Software Quality Properties

Functionality 2.1 might increase with increasing SIZE2.

Interoperability 2.1.3: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact.
Interoperability might increase with increasing SIZE2.

Security 2.1.4: Relating SIZE2 to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing SIZE2.

Reliability 2.2 might increase with increasing SIZE2.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by interface size.
Maturity might decline with increasing SIZE2.

Fault-tolerance 2.2.2: Relating SIZE2 to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing SIZE2.

Recoverability 2.2.3: Relating SIZE2 to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing SIZE2.

Efficiency 2.5 might decline with increasing SIZE2.

Time Behavior 2.5.1: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing SIZE2.

Resource Utilization 2.5.2: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing SIZE2.

Maintainability 2.6 declines with increasing SIZE2.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size. Stability might decline with increasing SIZE2.

References

- The SIZE2 metric is extensively discussed in [14, 2, 4, 11],
- SIZE2 is implemented in the VizzAnalyzer Metrics Suite.

Number Of local Methods (NOM)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the NOM specific view).

Handle NOM

Description Number of local Methods measures the number of methods locally declared in a class. Inherited methods are not considered. It is the size of the interface of a class and allows conclusions on its complexity.

Scope Class

View $V^{NOM} = (G^{NOM}, R^{NOM})$

- Grammar $G^{NOM} = (\{\text{class}^{NOM}, \text{method}^{NOM}\}, \emptyset, \text{class}^{NOM})$
- Relations $R^{NOM} : \{\text{methodof}^{NOM} : \text{method}^{NOM} \times \text{class}^{NOM}\}$
- Mapping α^{NOM}:

$$\begin{aligned} \alpha^{NOM}(\text{Class}) &\mapsto \text{class}^{NOM} \\ \alpha^{NOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{NOM} \\ \alpha^{NOM}(\text{Method}) &\mapsto \text{method}^{NOM} \end{aligned}$$

Definition The NOM value of a class $c \in class^{NOM}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{NOM}) \\ &\quad \textit{-- set of methods directly contained in } c \\ NOM(c) &= |M(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by NOM.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on its size.
Understandability declines with increasing NOM.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the size and complexity of its interface. NOM measures interface size.
Learnability might decline with increasing NOM.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends the complexity of its interface. NOM measures interface size.
Programmability might decline with increasing NOM.

Attractiveness 2.4.4: Attractiveness of a class depends on the size of the potentially reused code.
Attractiveness increases with increasing NOM.

Maintainability 2.6 declines with increasing NOM.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing NOM.

Changeability 2.6.2: Changing a class requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing NOM.

Testability 2.6.4: Complete testing requires coverage of all execution paths. The number of possible execution paths of a system increases with its size.
Testability declines with increasing NOM.

Portability 2.7 declines with increasing NOM.

Adaptability 2.7.1: As for changeability 2.6.2, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems.

Adaptability declines with increasing NOM.

Replaceablity 2.7.4: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by NOM.
Replaceablity decline with increasing NOM.

Related Software Quality Properties

Functionality 2.1 might increase with increasing NOM.

Interoperability 2.1.3: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact.
Interoperability might increase with increasing NOM.

Security 2.1.4: Relating NOM to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing NOM.

Reliability 2.2 might increase with increasing NOM.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by interface size.
Maturity might decline with increasing NOM.

Fault-tolerance 2.2.2: Relating NOM to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing NOM.

Recoverability 2.2.3: Relating NOM to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing NOM.

Efficiency 2.5 might decline with increasing NOM.

Time Behavior 2.5.1: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing NOM.

Resource Utilization 2.5.2: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing NOM.

Maintainability 2.6 declines with increasing NOM.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size.
Stability might decline with increasing NOM.

References

- The NOM metric is extensively discussed in [14, 2, 4, 11],
- and evaluated in a case study [19],
- NOM is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

3.1.3 Structural Complexity

McCabe Cyclomatic Complexity (CC)

CC can be computed on instances of a common meta-model, as long as the required types are provided by the front-end. A UML front-end, e.g., would not construct nodes of required type. Still, this front-end works with the new meta-model and the metric is unambiguously defined.

Description CC is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

Scope Method

View $V^{CC} = (G^{CC}, \emptyset)$.

- Grammar $G^{CC} = (T^{CC}, P^{CC}, \text{method}^{CC})$

- Entities $T^{CC} = \{\text{method}^{CC}, \text{ctrl\_stmt}^{CC}\}$
- Productions P^{CC}:

$$\begin{aligned} \text{method}^{CC} &= \text{ctrl\_stmt}^{CC\ *} \\ \text{ctrl\_stmt}^{CC} &= \text{ctrl\_stmt}^{CC\ *} \end{aligned}$$

- Mapping α^{CC}:

$$\begin{aligned} \alpha^{CC}(\text{Method}) &\mapsto \mathit{method}^{CC} \\ \alpha^{CC}(\text{Switch}) &\mapsto \text{ctrl\_stmt}^{CC} \\ \alpha^{CC}(\text{Loop}) &\mapsto \text{ctrl\_stmt}^{CC} \end{aligned}$$

Definition The CC value of a method $m \in \mathit{method}^{CC}$ is defined as:

$$CC(m) := \left|\mathit{succ}^{+}\left(m, \text{contains}^{CC}\right)\right| + 1$$

Scale Absolute.

Domain Integers in $1..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by attributed assess with Cyclomatic Complexity.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on its complexity.
Understandability declines with increasing Cyclomatic Complexity.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the complexity of its interface. Systems with high control complexity may also have a complex (behavioral) interface.
Learnability declines with increasing Cyclomatic Complexity.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends the complexity of its interface. Systems with high control complexity may also have a complex (behavioral) interface.
Programmability declines with increasing Cyclomatic Complexity.

Attractiveness 2.4.4: Attractiveness of a class depends on the complexity of the potentially reused code. Cyclomatic Complexity allows an assessment complexity.
Attractiveness increases with increasing Cyclomatic Complexity.

Maintainability 2.6 declines with increasing Cyclomatic Complexity.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified is directly related to the number of execution paths, i.e. the complexity of the control flow.
Analyzability declines with increasing Cyclomatic Complexity.

Changeability 2.6.2: Each modification must be correct for all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Changeability declines with increasing Cyclomatic Complexity.

Testability 2.6.4: Complete testing requires coverage of all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Testability declines with increasing Cyclomatic Complexity.

Portability 2.7 declines with increasing Cyclomatic Complexity.

Adaptability 2.7.1: As for changeability 2.6.2, the complexity the control structure of software has a direct impact. Each modification must be correct for all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Adaptability declines with increasing Cyclomatic Complexity.

Replaceablity 2.7.4: The substitute of a component must imitate its observable behavior. Components with complex control structures might have a more complex observable behavior making it more difficult to check substitutability and to

actually substitute a component.
Replaceablity declines with increasing Cyclomatic Complexity.

Related Software Quality Properties

Functionality 2.1 might increase with increasing Cyclomatic Complexity.

Interoperability 2.1.3: Relating Cyclomatic Complexity to interoperability requires to be able to locate the parts of a system responsible for interoperability. Complexity in these parts might indicate a better ability to interact.
Interoperability might increase with increasing CC.

Security 2.1.4: Relating Cyclomatic Complexity to security requires to be able to locate the parts of a system responsible for security. Complexity in these parts might indicate a higher security.
Security might increase with increasing Cyclomatic Complexity.

Reliability 2.2 might increase with increasing Cyclomatic Complexity.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by control-flow complexity.
Maturity might decline with increasing Cyclomatic Complexity.

Fault-tolerance 2.2.2: Relating Cyclomatic Complexity to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. Complexity in these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing CC.

Recoverability 2.2.3: Relating Cyclomatic Complexity to recoverability requires to be able to locate the parts of a system responsible for recoverability. Complexity in these parts might indicate a higher recoverability.
Recoverability might increase with increasing CC.

Efficiency 2.5 might decline with increasing Cyclomatic Complexity.

Time Behavior 2.5.1: Static complexity might indicate a higher execution complexity.
Time behavior might get worse with increasing Cyclomatic Complexity.

Maintainability 2.6 declines with increasing Cyclomatic Complexity.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by control-flow complexity.
Stability might decline with increasing Cyclomatic Complexity.

References

- CC is extensively discussed in [22],
- it is used to assess design and structural complexity of a system in [18],
- Li and Henry [15] propose to use Cyclomatic Complexity 3.1.3 as weight measure in Weighted Method Count 3.1.3,
- it is mentioned in the following resources:
  - `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`
  - `http://www.mccabe.com/iq_research_metrics.htm`
  - `http://mdp.ivv.nasa.gov/repository.html`
- CC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Weighted Method Count (WMC)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the WMC specific view).

Handle WMC

Description A weighted sum of methods implemented within a class. It is parameterized by a way to compute the weight of each method. Possible weight metrics are:

- McCabe Cyclomatic Complexity 3.1.3,
- Lines of Code 3.1.1,
- 1 (unweighted WMC).

This variant of WMC uses McCabe Cyclomatic Complexity 3.1.3 metric for calculating the weight for each method. Originally defined as an object-oriented metric, it can easily adapted to non-object-oriented systems computing the weighted sum of functions implemented within a module or file.

Scope Class

View $V^{WMC} = (G^{WMC}, R^{WMC})$

- Grammar $G^{WMC} = (\{\text{class}^{WMC}, \text{method}^{WMC}\}, \emptyset, \text{class}^{WMC})$
- Relations $R^{WMC} : \{\text{methodof}^{WMC} : \text{method}^{WMC} \times \text{class}^{WMC}\}$
- Mapping α^{WMC}:

$$\begin{aligned} \alpha^{WMC}(\text{Class}) &\mapsto \text{class}^{WMC} \\ \alpha^{WMC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{WMC} \\ \alpha^{WMC}(\text{Method}) &\mapsto \text{method}^{WMC} \end{aligned}$$

Definition The WMC value of a class $c \in class^{WMC}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{WMC}) \\ &\quad \textit{-- set of methods directly contained in } c \\ WMC(c) &= \sum_{m \in M(c)} CC(m) \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 depends on the weight metric, which influences the attributes assessed with WMC negatively or positively.

Re-Usability declines with increasing unweighted WMC.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on the size of its interface.
Understandability declines with increasing unweighted WMC.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the size of its interface.
Learnability declines with increasing unweighted WMC.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends on the size of its interface.
Programmability declines with increasing unweighted WMC.

Attractiveness 2.4.4: Attractiveness of a class depends on the size and complexity of the potentially reused code. Depending on the weight metric, WMC allows an assessment of size or complexity.
Attractiveness increases with increasing unweighted WMC.

Maintainability 2.6 declines with increasing WMC.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of parts to be modified is directly related to the size and complexity of the class. Depending on the weight metric, WMC allows an assessment of size or complexity.
Analyzability declines with increasing unweighted WMC.

Changeability 2.6.2: Changing a class requires prior understanding, which, in turn, is more complicated for large and complex systems. Depending on the weight metric, WMC allows an assessment of size or complexity.
Changeability declines with increasing unweighted WMC.

Testability 2.6.4: Complete testing requires coverage of all execution paths. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. Depending on the weight metric, WMC allows an assessment of the number of methods and their complexity.

Testability declines with increasing unweighted WMC.

Portability 2.7 declines with increasing WMC.

Adaptability 2.7.1: As for changeability 2.6.2, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems. Size is specifically assessed by the weighted versions of WMC.
Adaptability declines with increasing unweighted WMC.

Replaceablity 2.7.4: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by the unweighted WMC.
Replaceablity decline with increasing unweighted WMC.

Related Software Quality Properties

Functionality 2.1 might increase with increasing WMC.

Interoperability 2.1.3: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact.
Interoperability might increase if the unweighted WMC increases.

Security 2.1.4: Relating WMC to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing unweighted WMC.

Reliability 2.2 might increase with increasing WMC.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by WMC.
Maturity might decline with increasing unweighted WMC.

Fault-tolerance 2.2.2: Relating WMC to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-tolerance might increase if the unweighted WMC increases.

Recoverability 2.2.3: Relating WMC to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase if the unweighted WMC increases.

Efficiency 2.5 might decline with increasing WMC.

Time Behavior 2.5.1: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing unweighted WMC.

Resource Utilization 2.5.2: Static size might indicate a higher memory utilization.
Resource utilization might get worse if the unweighted WMC increases.

Maintainability 2.6 declines with increasing WMC.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size.
Stability might decline with increasing unweighted WMC.

References

- WMC is extensively discussed in [5, 6, 14, 21, 3, 2, 1, 7, 11, 12, 17, 4, 10],
- Li and Henry [15] propose to weight the methods according to Cyclomatic Complexity 3.1.3,
- it is evaluated in a case study [19],
- WMC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Response For a Class (*RFC*)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective call (Java) or message (UML) relations expressing the messages sent between two

classes are stored in the CMM10 as Invokes, and are mapped onto relations of type call^{RFC} in the common meta-model (and the RFC specific view).

Handle RFC

Description Count of (public) methods in a class and methods directly called by these. RFC is only applicable to object-oriented systems.

Scope Class

View $V^{RFC} = (G^{RFC}, R^{RFC})$

- Grammar $G^{RFC} = (\{\text{class}^{RFC}, \text{method}^{RFC}\}, \emptyset, \text{class}^{RFC})$
- Relations $R^{RFC} : \{\text{contains}^{RFC} : \text{class}^{RFC} \times \text{method}^{RFC},$
  $\text{call}^{RFC} : \text{method}^{RFC} \times \text{method}^{RFC},$
  $\text{call}^{RFC} : \text{class}^{RFC} \times \text{method}^{RFC}\}$
- Mapping α^{RFC}:

$$\begin{aligned} \alpha^{RFC}(\text{Class}) &\mapsto \text{class}^{RFC} \\ \alpha^{RFC}(\text{Method}) &\mapsto \text{method}^{RFC} \\ \alpha^{RFC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{RFC} \\ \alpha^{RFC}(\text{Invokes}) &\mapsto \text{call}^{RFC} \end{aligned}$$

Definition The RFC value of a class $c \in class^{RFC}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{RFC}) \\ &\quad \textit{-- set of methods contained in } c \\ R(c) &= succ(M(c), \text{call}^{RFC}) \\ &\quad \textit{-- set of methods called by methods contained in } c \\ RFC(c) &= |M(c) \cup R(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by attributed assessed with *Response For a Class*.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on its complexity and size of the method set it is related to.
Understandability declines with increasing RFC.

Attractiveness 2.4.4: Attractiveness of a class depends on the complexity of the potentially reused code. Response For a Class allows an assessment of complexity.
Attractiveness increases with increasing RFC.

Maintainability 2.6 declines with increasing response set size.

Analysability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified is directly related to the number of executed methods in response to a message.
Analysability declines with increasing RFC.

Changeability 2.6.2: Each modification must be correct for all execution paths. The size of the response set for a class (RFC) gives an idea about how many methods are potentially contributing to the size of the execution paths.
Changeability declines with increasing RFC.

Testability 2.6.4: Complete testing requires coverage of all execution paths. Response For a Class computes the number of methods (directly) involved in handling a particular message.
Testability declines with increasing RFC.

Portability 2.7 declines with increasing response set size.

Adaptability 2.7.1: As for changeability 2.6.2, the complexity the control structure of software has a direct impact. Each modification must be correct for all execution paths. The size of the response set for a class (RFC) gives an idea about how many methods are potentially contributing to the size of the execution paths.
Adaptability declines with increasing RFC.

Related Software Quality Properties

Reliability 2.2 might decrease with increasing response set size.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by response set size.
Maturity might decline with increasing RFC.

Re-Usability 2.4 is both negatively and positively influenced by attributed assess with Response For a Class.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the complexity of its interface (public methods) and the number of methods in other classes called in response to a received message.
Learnability declines with increasing RFC.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends the complexity of its interface and dependency on other classes (implementing not local methods).
Programmability declines with increasing RFC.

Efficiency 2.5 might decline with increasing response set size.

Time Behavior 2.5.1: Static complexity might indicate a higher execution complexity.
Time behavior might get worse with increasing RFC.

Maintainability 2.6 declines with increasing response set size.

Stability 2.6.3: Due to reduced analysability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by response set size.
Stability might decline with increasing RFC.

Portability 2.7 declines with increasing response set size.

Replaceablity 2.7.4: The substitute of a component must imitate its observable behavior. Components with complex control structures and response sets might have a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component.
Replaceablity declines with increasing RFC.

References

- RFC is extensively discussed and validated in [5, 6, 3, 14, 2, 1, 7, 11, 12, 10, 4, 21, 17],
- RFC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

3.2 Architecture and Structure

3.2.1 Inheritance

Depth of Inheritance Tree (DIT)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the DIT specific view).

Handle DIT

Description Depth of Inheritance Tree (DIT) is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class. DIT is only applicable to object-oriented systems.

Scope Class

View $V^{DIT} = (G^{DIT}, R^{DIT})$

- Grammar $G^{DIT} = (\{\text{class}^{DIT}\}, \emptyset, \text{class}^{DIT})$
- Relations $R^{DIT} : \{\text{inheritance}^{DIT} : \text{class}^{DIT} \times \text{class}^{DIT}\}$
- Mapping α^{DIT}:

$$\begin{aligned} \alpha^{DIT}(\text{Class}) &\mapsto \text{class}^{DIT} \\ \alpha^{DIT}(\text{Inheritance}) &\mapsto \text{inheritance}^{DIT} \end{aligned}$$

Definition The DIT value of a class $c \in class^{DIT}$ is defined as:

$$\begin{aligned} P(c) &= pred^{*}(c, \text{inheritance}^{DIT}) \\ &\quad \textit{-- set of classes, c inherits from directly or indirectly} \\ DIT(c) &= max(dist(c, P(c))) \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is both negatively and positively influenced by attributed assess with DIT.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Understandability declines with increasing DIT.

Learnability for Reuse 2.4.2: Learning how to use a class depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Learnability declines with increasing DIT.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Programmability declines with increasing DIT.

Attractiveness 2.4.4: Attractiveness of a class depends on the size of the potentially reused code. Classes that are deep down

in the classes hierarchy potentially inherit many methods from super-classes.
Attractiveness increases with increasing DIT.

Maintainability 2.6 declines with increasing DIT.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of parts to be modified is related to the number of methods of the class. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Analyzability declines with increasing DIT.

Changeability 2.6.2: Changing a class requires prior understanding, which, in turn, is more complicated for classes with many methods. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to understand it.
Changeability declines with increasing DIT.

Testability 2.6.4: Complete testing requires coverage of all execution paths. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Due to late binding, super-class methods need to be tested again in the sub-class context. This makes it it potentially harder to test classes deep down in the classes hierarchy.
Testability declines with increasing DIT.

Portability 2.7 declines with increasing DIT.

Adaptability 2.7.1: As for changeability 2.6.2, the size of software has a direct impact. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Adaptability declines with increasing DIT.

Replaceablity 2.7.4: The substitute of a component must im-

itate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size increases for classes that are deep down in the classes hierarchy. Replaceablity decline with increasing DIT.

Related Software Quality Properties

Functionality 2.1 might increase with increasing DIT.

Interoperability 2.1.3: Relating DIT to interoperability requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a better ability to interact.
Interoperability might increase with increasing DIT.

Security 2.1.4: Relating DIT to security requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a higher security.
Security might increase with increasing DIT.

Reliability 2.2 is both positively and negatively influenced by attributes assessed with DIT.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by DIT.
Maturity might decline with increasing DIT.

Fault-tolerance 2.2.2: Relating DIT to fault-tolerance requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a better ability to interact.
Fault-Tolerance might increase with increasing DIT.

Recoverability 2.2.3: Relating DIT to recoverability requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a higher recoverability.
Recoverability might increase with increasing DIT.

Efficiency 2.5 might decline with increasing DIT.

Time Behavior 2.5.1: Static size might indicate a higher execution time due to increased number of instruction cache misses,

long jumps, etc. Classes inheriting many attributes are potentially large. Moreover, late binding requires indirect calls and prevents optimizations in the context of the caller.
Time behavior might get worse with increasing DIT.

Resource Utilization 2.5.2: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing DIT.

Maintainability 2.6 declines with increasing DIT.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size.
Stability might decline with increasing DIT.

References

- DIT is extensively discussed and evaluated in [5, 6, 3, 14, 2, 1, 7, 11, 12, 10, 4, 21, 17, 19],
- DIT is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Number Of Children (NOC)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the NOC specific view).

Handle NOC

Description NOC is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. NOC measures how many classes inherit directly methods or fields from a super-class. NOC is only applicable to object-oriented systems.

Scope Class

View $V^{NOC} = (G^{NOC}, R^{NOC})$

- Grammar $G^{NOC} = (\{\text{class}^{NOC}\}, \emptyset, \text{class}^{NOC})$

- Relations R^{NOC} : $\{\text{inheritance}^{NOC} : \text{class}^{NOC} \times \text{class}^{NOC}\}$
- Mapping α^{NOC}:

$$\begin{aligned} \alpha^{NOC}(\text{Class}) &\mapsto \text{class}^{NOC} \\ \alpha^{NOC}(\text{Inheritance}) &\mapsto \text{inheritance}^{NOC} \end{aligned}$$

Definition The NOC value of a class $c \in class^{NOC}$ is defined as:

$$NOC(c) := \left|succ(c, \text{inheritance}^{NOC})\right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is positively influenced by attributes assessed with NOC.

Understandability for Reuse 2.4.1: Understanding a class is supported if it has a high number of children. First numerous children show directly a high re-use (re-usability) of the class under concern. Further with numerous children exist many examples on how to re-use the class, which helps to understand weather a class is suitable, and how it can be re-used for a particular tasks.
Understandability increases with increasing NOC.

Learnability for Reuse 2.4.1: Learning (acquiring and memorizing knowledge) how to (re-)use a class by extending it is influenced by the number of children the class under concern has. A high number of children, that is many re-using child classes, allows acquiring versatile knowledge on how to re-use a particular class in different situations. This supports through redundancy memorizing the knowledge about how to re-use a particular class.
Learnability increases with a increasing NOC.

Operability for Reuse 2.4.1: Understanding if a class can be integrated and controlled by software engineers/developers depends on the number of children a class has. A high number

of children indicates that a particular class is well integrated into an existing software system. This means, that it is suitable for several different tasks, self-descriptive since there exist many examples on its usage, having a higher error tolerance, since it is involved each time one of its children is tested.
Operability increases with increasing NOC.

Attractiveness for Reuse 2.4.1: A class having a higher number of children appears more attractive to the software engineer/developer. It is a eye-catcher in a class diagram or other representations since it stands out by having many children. Associated assumptions could be, that the class is stable, since it is tested each time a child is tested, that is well documented and understood, since it has been extended so often, that it is easier to understand, since there are many examples of usage, that it helps to understand the children, if the parent class has been understood, that is plays a central role in the design, since many classes extend its functionality.
Attractiveness increases with increasing NOC.

Maintainability 2.6 is negatively and positively influenced by attributes assessed with NOC.

Changeability 2.6: A class having a higher number of children has a lower changeability. The effort spent on modification, fault removal or environmental change is increased, since many child classes are extending its functionality, depending on the parent class. The side effects of modifications are harder to predict. Fault removal effects child classes. The need for environmental change has to consider the child classes.
Changeability decreases with increasing NOC.

Portability 2.7 is negatively influenced by attributes assessed with NOC.

Replaceability 2.7: A class having a higher number of children is difficult to replace. The children are dependent on it, by extending specific functionality and can depend on certain functionality the parent class provides. It is difficult to find another class satisfying these specific needs, allowing to replace the parent class.
Replaceability decreases with increasing NOC.

Related Software Quality Properties

Reliability 2.2 is positively influenced by attributes assessed with *NOC*.

Maturity 2.2.1: A class having a higher number of children has a high maturity. The frequency of failure by faults is low, since the many faults have been identified in the various ways the child classes interact with the parents. The parent class is heavily used, since it is used each time a child class is used. Maturity increases with increasing NOC.

Maintainability 2.6 is positively and negatively influenced by attributes assessed with NOC.

Analysability 2.6.1: To analyze a class having a higher number of children requires higher effort. Diagnosis of deficiencies or causes of failures involves the children of a parent class. They have access to the functionality and data provided by a class and have to be involved in the analysis. Identifying parts in a parent class which need to be modified requires the analysis of all child classes, which are effected by the modification. To analyze a class completely it is also necessary to look at its children to be able to have the complete picture.
Analysability decreases with increasing NOC.

Stability 2.6.3: A class having a higher number of children bears a higher risk of unexpected effect of modification. Child classes are re-using parent classes in various ways. They are directly effected by modifications making it difficult to predict how stable a class or software product would be after modification.
Stability decreases with increasing NOC.

Testability 2.6.4: A class having a higher number of children requires a higher effort for validating the modified software. All depending child classes need to be included in the tests as well, since modifications in the parent class have direct impact on the extending child classes. The amount of designed and implemented autonomous test aid functions and test complexity is increased.
Testability decreases with increasing NOC.

Portability 2.7 is negatively influenced by attributes assessed with NOC.

Adaptability 2.7.1: A class having a higher number of children is difficult adapt to different specified environments. The children are dependent on it, by extending specific functionality and can depend on certain functionality the parent class provides. Adapting the parent class to a new environment, can make it unsuitable for the children, requiring adaptation in them as well.
Adaptability decreases with increasing NOC.

References

- NOC is extensively discussed and evaluated in [5, 6, 7, 3, 14, 2, 1, 11, 12, 10, 4, 21, 17, 19],
- NOC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

3.2.2 Coupling

Afferent Coupling (Ca)

Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Afferent Coupling between packages (Ca) measures the total number of external classes coupled to classes of a package due to incoming coupling (coupling from classes external classes of the package, uses CBO definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes do not

use the package's classes. Ca is primarily applicable to object-oriented systems.

Scope Package

View $V^{Ca} = (G^{Ca}, R^{Ca})$

- Grammar $G^{Ca} = (\{\text{package}^{Ca}, \text{class}^{Ca}\}, \emptyset, \text{package}^{Ca})$
- Relations $R^{Ca} = \{\text{coupling}^{Ca} : \text{class}^{Ca} \times \text{class}^{Ca}\}$
- Mapping α^{Ca}:

$$\begin{aligned} \alpha^{Ca}(\text{Class}) &\mapsto \text{class}^{Ca} \\ \alpha^{Ca}(\text{Package}) &\mapsto \text{package}^{Ca} \\ \alpha^{Ca}(\text{Invokes}) &\mapsto \text{coupling}^{Ca} \\ \alpha^{Ca}(\text{Accesses}) &\mapsto \text{coupling}^{Ca} \\ \alpha^{Ca}(\text{IsOfType}) &\mapsto \text{coupling}^{Ca} \end{aligned}$$

Definition The Ca value of a package $p \in \text{package}^{Ca}$ is defined:

$$\begin{aligned} CIP(p) &= succ^*(p, \text{contains}^{Ca}) \\ &\quad \textit{-- set of classes inside/contained in } p \\ COP(p) &= \{c \in \text{class}^{Ca} | c \notin CIP(p)\} \\ &\quad \textit{-- set of classes outside } p \\ Coupled(p) &= \{c \in \text{class}^{Ca} | c \in pred(CIP(p), \text{coupling}^{Ca}) \wedge \\ &\quad c \in COP(p)\} \\ &\quad \textit{-- set of classes coupled to } p \textit{ over afferent coupling} \\ Ca(p) &= |Coupled(p)| \end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties

Portability 2.7 is negatively influenced by attributes assessed with Ca.

Replaceability 2.7.4 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be highly inversely related to replaceability, since other parts depend on it. Replaceability decreases with increasing Ca.

Related Software Quality Properties

Functionality 2.1 is positively and negatively influenced by attributes assessed with Ca.

Interoperability 2.1.3 Parts of a system showing a high afferent (ingoing) coupling to other system parts may be directly related to interoperability, since they are used/interacted with from other parts of the system.
Interoperability decreases with increasing Ca.

Security 2.1.4 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to security, since they can be influenced in many ways from other parts of the system.
Security decreases with increasing Ca.

Reliability 2.2 is negatively influenced by attributes assessed with Ca.

Fault-tolerance 2.2.2 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to fault-tolerance, since a local fault might be propagated to other parts of the system.
Fault-tolerance decreases with increasing Ca.

Re-Usability 2.4.1 is positively influenced by attributes assessed with Ca.

Learnability 2.3.2 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be directly related to learnability, since other parts of the system using them serve as examples.
Learnability increases with increasing Ca.

Operability for Reuse – Programmability 2.4.3 The part of a system that has a high afferent (ingoing) coupling from other system parts may be directly related to programmability, since other parts of the system using it serve as examples.

Operability for Reuse – Programmability increases with increasing Ca.

Attractiveness 2.4.4 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be directly related to attractiveness, since other parts of the system use them might show a good re-usability.
Attractiveness increases with increasing Ca.

Maintainability 2.6 is negatively influenced by attributes assessed with Ca.

Stability 2.6.3 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to stability, since other parts are affected by them.
Stability decreases with increasing Ca.

References

- Ca is discussed in [16],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Coupling Between Objects (CBO)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the CBO specific view).

Handle CBO

Description Coupling Between Objects (CBO) is the number of other classes that a class is coupled to. CBO is only applicable to object-oriented systems.

Scope Class

View $V^{CBO} = (G^{CBO}, R^{CBO})$

- Grammar $G^{CBO} = (\{\text{class}^{CBO}, \text{method}^{CBO}\}, \emptyset, \text{class}^{CBO})$

- Relations $R^{CBO} = \{\text{coupling}^{CBO} : \text{class}^{CBO} \times \text{class}^{CBO}\}$
- Mapping α^{CBO}:

$$\begin{aligned}
\alpha^{CBO}(\text{Class}) &\mapsto \text{class}^{CBO} \\
\alpha^{CBO}(\text{IsDefinedInTermsOf}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{IsParameterOf}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{InheritsFrom}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{IsOfType}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{Accessess}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{Invokes}) &\mapsto \text{coupling}^{CBO}
\end{aligned}$$

Definition The CBO value of a class $c \in class^{CBO}$ is defined as:

$$CBO(c) = \left|succ(c, \text{coupling}^{CBO}) \backslash c\right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts, which have a high (outgoing) efferent coupling may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CBO.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CBO.

Maintainability 2.6 decreases with increasing CBO.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since

they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CBO.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CBO.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CBO.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CBO.

Portability 2.7 decreases with increasing CBO.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CBO.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CBO.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CBO.

Reliability 2.2 might decrease with increasing CBO.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CBO.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CBO.

Re-Usability 2.4 might decrease with increasing CBO.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CBO.

Operability for Reuse – Programmability 2.4.3: Parts having a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CBO.

Efficiency 2.5 might decrease with increasing CBO.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CBO.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CBO.

References

- CBO is extensively discussed and validated in [5, 6, 7, 3, 14, 2, 1, 11, 12, 10, 4, 21, 17, 19],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Change Dependency Between Classes ($CDBC$)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $CDBC$ specific view).

Handle $CDBC$

Description The Change Dependency Between Classes ($CDBC$) measures the class level coupling. It is a measure assigned to pairs of classes describing how dependent one class (client class) is on the other (server class). This allows conclusions on the follow-up work to be done in a client class, when the server class is changed in the course of re-engineering.

Scope Class pairs

View $V^{CDBC} = (G^{CDBC}, R^{CDBC})$

- Grammar $G^{CDBC} = (\{\text{class}^{CDBC}, \text{method}^{CDBC}\}, \emptyset, \text{class}^{CDBC})$
- Relations $R^{CDBC} = \{\text{call}^{CDBC} : \text{method}^{CDBC} \times \text{method}^{CDBC}$,
  $\text{access}^{CDBC} : \text{method}^{CDBC} \times \text{class}^{CDBC}$,
  $\text{containsMethod}^{CDBC} : \text{class}^{CDBC} \times \text{method}^{CDBC}$,
  $\text{inherits}^{CDBC} : \text{class}^{CDBC} \times \text{class}^{CDBC}\}$

- Mapping α^{CDBC}:

$$\begin{aligned}
\alpha^{CDBC}(\text{Class}) &\mapsto \text{class}^{CDBC} \\
\alpha^{CDBC}(\text{Method}) &\mapsto \text{method}^{CDBC} \\
\alpha^{CDBC}(\text{IsMethodOf}) &\mapsto \text{containsMethod}^{CDBC} \\
\alpha^{CDBC}(\text{InheritsFrom}) &\mapsto \text{inherits}^{CDBC} \\
\alpha^{CDBC}(\text{IsOfType}) &\mapsto \text{typeRef}^{CDBC} \\
\alpha^{CDBC}(\text{Accessess}) &\mapsto \text{access}^{CDBC} \\
\alpha^{CDBC}(\text{Invokes}) &\mapsto \text{call}^{CDBC}
\end{aligned}$$

Definition The $CDBC$ value of a pair of classes $CC, SC \in class^{CDBC}$ is defined as:

$$\begin{aligned}
CDBC(CC, SC) &= min\,(n, A) \\
&\quad \textit{-- with } n \textit{ is the number of methods} \\
A &= \sum_{impl=1}^{m_1} \alpha_{impl} + (1-k) \sum_{interface=1}^{m_2} \alpha_{interface} \\
&\quad \textit{-- with } m_1 \textit{ are the methods in } CC \textit{ accessing} \\
&\quad \textit{the implementation of } SC \textit{ being possibly} \\
&\quad \textit{affected} \\
&\quad \textit{-- with } m_2 \textit{ are the methods in } CC \textit{ accessing} \\
&\quad \textit{the interface of } SC \textit{ being possibly affected}
\end{aligned}$$

α is determined according to the following table:

Scale Ordinal.

Domain Integers in $0..\infty$, 0 means there is no change dependency between the pair of classes.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts, which have a high (outgoing) efferent coupling may be highly inversely related

Relationship types between CC and SC	α = number of methods of CC potentially affected by a change
SC is not used by CC at all	0
SC is the class of an instance variable of CC	n
Local variables of type SC are used within j methods of CC	j
SC is a superclass of CC	n
SC is the parameter type of j methods of CC	j
CC accesses a global variable of class SC	n

Table 3.1: Relationship types between CC and SC and their corresponding contribution α to change dependency.

to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CDBC.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CDBC.

Maintainability 2.6 decreases with increasing CDBC.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CDBC.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CDBC.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CDBC.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they

are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CDBC.

Portability 2.7 decreases with increasing CDBC.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CDBC.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CDBC.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CDBC.

Reliability 2.2 might decrease with increasing CDBC.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CDBC.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CDBC.

Re-Usability 2.4 might decrease with increasing CDBC.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability,

since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CDBC.

Operability for Reuse – Programmability 2.4.3: Parts having a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CDBC.

Efficiency 2.5 might decrease with increasing CDBC.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CDBC.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CDBC.

References

- CDBC is extensively discussed and evaluated in [11, 12, 2, 19],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Change Dependency Of Classes (*CDOC*)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *CDOC* specific view).

Handle *CDOC*

Description The Change Dependency Of Classes ($CDOC$) measures the class level coupling. It is a measure assigned to classes describing how dependent other classes (client classes) are on this class (server class). This allows conclusions on the follow-up work to be done in all client class, when the server class is changed in the course of re-engineering. It is a accumulation of the $CDBC$ metric, for a server class (SC) and all its client classes (CC).

Scope Class

View $V^{CDOC} = (G^{CDOC}, R^{CDOC})$

- Grammar $G^{CDOC} = (\{\text{class}^{CDOC}\}, \emptyset, \text{class}^{CDOC})$
- Relations $R^{CDOC} = \{\emptyset\}$
- Mapping α^{CDOC}:

$$\alpha^{CDOC}(\text{Class}) \quad \mapsto \quad \text{class}^{CDOC}$$

Definition The $CDOC$ value of a class $c \in class^{CDOC}$ is defined as:

$$CDOC(c) \quad = \quad \sum_{s \in class^{CDOC} \setminus c} CDBC(s, c)$$

Scale Absolute.

Domain Integers in $0..\infty$, 0 means there is no change dependency between the class to any other class in the system.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts having a high (outgoing) efferent coupling may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CDOC.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CDOC.

Maintainability 2.6 decreases with increasing CDOC.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CDOC.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CDOC.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CDOC.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CDOC.

Portability 2.7 decreases with increasing CDOC.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CDOC.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CDOC.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CDOC.

Reliability 2.2 might decrease with increasing CDOC.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CDOC.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CDOC.

Re-Usability 2.4 might decrease with increasing CDOC.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CDOC.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CDOC.

Efficiency 2.5 might decrease with increasing CDOC.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CDOC.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CDOC.

References

- CDOC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Efferent Coupling (Ce)

Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Efferent Coupling between packages (Ce) measures the total number of external classes coupled to classes of a package due to outgoing coupling (coupling to classes external classes of the package, uses Ce definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. Ce is primarily applicable to object-oriented systems.

Scope Package

View $V^{Ce} = (G^{Ce}, R^{Ce})$

- Grammar $G^{Ce} = (\{\text{package}^{Ce}, \text{class}^{Ce}\}, \emptyset, \text{package}^{Ce})$
- Relations $R^{Ce} = \{\text{coupling}^{Ce} : \text{class}^{Ce} \times \text{class}^{Ce}\}$

- Mapping α^{Ce}:

$$\begin{aligned}
\alpha^{Ce}(\text{Class}) &\mapsto \text{class}^{Ce} \\
\alpha^{Ce}(\text{Package}) &\mapsto \text{package}^{Ce} \\
\alpha^{Ce}(\text{Invokes}) &\mapsto \text{coupling}^{Ce} \\
\alpha^{Ce}(\text{Accesses}) &\mapsto \text{coupling}^{Ce} \\
\alpha^{Ce}(\text{IsOfType}) &\mapsto \text{coupling}^{Ce}
\end{aligned}$$

Definition The Ce value of a package $p \in \text{package}^{Ce}$ is defined:

$$\begin{aligned}
CIP(p) &= succ^*(p, \text{contains}^{Ce}) \\
&\quad \textit{-- set of classes inside/contained in } p \\
COP(p) &= \{c \in \text{class}^{Ce} | c \notin CIP(p)\} \\
&\quad \textit{-- set of classes outside } p \\
Coupled(p) &= \{c \in \text{class}^{Ce} | c \in succ(CIP(p), \text{coupling}^{Ce}) \wedge \\
&\quad c \in COP(p)\} \\
&\quad \textit{-- set of classes coupled to } p \textit{ over afferent coupling} \\
Ce(p) &= |Coupled(p)|
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties :

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well. Understandability decreases with increasing Ce.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies. Attractiveness decreases with increasing Ce.

Maintainability 2.6 decreases with increasing Ce.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing Ce.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing Ce.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing Ce.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing Ce.

Portability 2.7 decreases with increasing Ce.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing Ce.

Related Software Quality Properties :

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing Ce.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing Ce.

Reliability 2.2 might decrease with increasing Ce.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing Ce.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing Ce.

Re-Usability 2.4 might decrease with increasing Ce.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing Ce.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing Ce.

Efficiency 2.5 might decrease with increasing Ce.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing Ce.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus exe-

cution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing Ce.

References

- Ce is discussed in [16],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Coupling Factor (CF)

Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Coupling Factor (CF) measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes. CF is primarily applicable to object-oriented systems.

Scope Package

View $V^{CF} = (G^{CF}, R^{CF})$

- Grammar $G^{CF} = (\{\text{package}^{CF}, \text{class}^{CF}\}, P^{CF}, \text{package}^{CF})$
- Productions $P^{CF} = \{\text{package}^{CF} ::= \text{class}^{CF\ *}\}$
- Relations $R^{CF} : \{\text{coupling}^{CF}\}$[1]

[1]The structural contains^{CF} relation is implicitly defined by the productions P^{CF}.

- Mapping α^{CF}:

$$\begin{aligned}
\alpha^{CF}(\text{Class}) &\mapsto \text{class}^{CF} \\
\alpha^{CF}(\text{Package}) &\mapsto \text{package}^{CF} \\
\alpha^{CF}(\text{Invokes}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{Accesses}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{IsOfType}) &\mapsto \text{coupling}^{CF}
\end{aligned}$$

Definition The CF value of a package $p \in \text{package}^{CF}$ is defined:

$$\begin{aligned}
Classes(p) &= succ^*(p, \text{contains}^{CF}) \cap \text{class}^{\texttt{CF}} \\
&\quad \textit{-- set of classes contained in } p \\
Coupled(p,c) &= succ(c, \text{coupling}^{CF}) \cap Classes(p) \\
&\quad \textit{-- set of classes contained in } p, \\
&\quad \textit{-- which } c \textit{ is coupled to} \\
CF(p) &= \frac{\sum_{c \in Classes(p)} \mid Coupled(p,c) \mid}{0.5 * \mid Classes(p) \mid^2 - \mid Classes(p) \mid}
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A art of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well. Understandability decreases with increasing CF.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies. Attractiveness decreases with increasing CF.

Maintainability 2.6 decreases with increasing CF.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CF.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CF.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CF.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CF.

Portability 2.7 decreases with increasing CF.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing CF.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CF.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.

Security might decrease with increasing CF.

Reliability 2.2 might decrease with increasing CF.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CF.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CF.

Re-Usability 2.4 might decrease with increasing CF.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CF.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CF.

Efficiency 2.5 might decrease with increasing CF.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CF.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CF.

References

- CF is discussed in [8, 2, 17, 9],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Data Abstraction Coupling (DAC)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the DAC specific view).

Handle DAC

Description The DAC measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes. Basically same as DAC, but coupling limited to type references.

Scope Class

View $V^{DAC} = (G^{DAC}, R^{DAC})$

- Grammar $G^{DAC} = (\{\text{class}^{DAC}\}, \emptyset, \text{class}^{DAC})$
- Relations $R^{DAC} : \{\text{referencestype}^{DAC} : \text{class}^{DAC} \times \text{class}^{DAC}\}$
- Mapping α^{DAC}:

$$\begin{aligned} \alpha^{DAC}(\text{Class}) &\mapsto \text{class}^{DAC} \\ \alpha^{DAC}(\text{IsOfType}) &\mapsto \text{referencestype}^{DAC} \end{aligned}$$

Definition The DAC value of a class $c \in class^{DAC}$ is defined as:

$$DAC(c) = \left| succ(c, \text{referencestype}^{DAC}) \backslash c \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it is uses other parts of the system which need to be understood as well.
Understandability decreases with increasing DAC.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing DAC.

Maintainability 2.6 decreases with increasing DAC.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing DAC.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing DAC.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing DAC.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing DAC.

Portability 2.7 decreases with increasing DAC.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing DAC.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing DAC.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing DAC.

Reliability 2.2 might decrease with increasing DAC.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing DAC.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing DAC.

Re-Usability 2.4 might decrease with increasing DAC.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing DAC.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing DAC.

Efficiency 2.5 might decrease with increasing DAC.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing DAC.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing DAC.

References

- DAC is extensively discussed and validated in [14, 2, 4, 11, 19],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Instability (I)

Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. It does not relay on relations since it uses existing metric values for its calculation. By defining a view containing only packages as elements, the metric definition can ignore classes, methods and fields as part of its description, since the relations originating from them are not relevant.

Description Instability between packages (I) measures the ratio between the outgoing and the total number of in- and outgoing couplings from classes inside the package from/to classes outside the package (coupling

to classes external classes of the package, uses I definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. I is primarily applicable to object-oriented systems.

Scope Package

View $V^I = (G^I, R^I)$

- Grammar $G^I = (\{\text{package}^I\}, \emptyset, \text{package}^I)$
- Relations $R^I = \{\emptyset\}$
- Mapping α^I:

$$\alpha^I(\text{Package}) \quad \mapsto \quad \text{package}^I$$

Definition The I value of a package $p \in \text{package}^I$ is defined:

$$I(p) \quad = \quad \frac{p.Ce}{p.Ca + p.Ce}$$

Scale Absolute.

Domain Rationale in $0..1.0\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing I.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing I.

Maintainability 2.6 decreases with increasing I.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing I.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing I.

Stability 2.6.3: Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to stability, since other parts are affected by them. Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing I.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing I.

Portability 2.7 decreases with increasing I.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing I.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.

Interoperability might increase with increasing I.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing I.

Reliability 2.2 might decrease with increasing I.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing I.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing I.

Re-Usability 2.4 might decrease with increasing I.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing I.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing I.

Efficiency 2.5 might decrease with increasing I.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing I.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utiliza-

tion, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing I.

References

- I is discussed in [16],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Locality of Data (LD)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the LD specific view).

Handle LD

Description The Locality of Data metric relates the amount of data being local the class to the total amount of data used by the class. This relates to the quality of abstraction embodied by the class and allows conclusions on the reuse potential of the class and testability.

Scope Class

View $V^{LD} = (G^{LD}, R^{LD})$

- Grammar $G^{LD} = (\{\text{class}^{LD}, \text{data}^{LD}\}, \emptyset, \text{class}^{LD})$
- Relations $R^{LD} : \{\text{uses}^{LD} : \text{method}^{LD} \times \text{data}^{LD},$
  $R^{LD} : \{\text{containeddata}^{LD} : \text{class}^{LD} \times \text{data}^{LD},$
  $R^{LD} : \{\text{containedmethods}^{LD} : \text{class}^{LD} \times \text{method}^{LD}\}$

- Mapping α^{LD}:

$$
\begin{aligned}
\alpha^{LD}(\text{Class}) &\mapsto \text{class}^{LD} \\
\alpha^{LD}(\text{AVariable}) &\mapsto \text{data}^{LD} \\
\alpha^{LD}(\text{Method}) &\mapsto \text{method}^{LD} \\
\alpha^{LD}(\text{IsFieldOf}) &\mapsto \text{isdataof}^{LD} \\
\alpha^{LD}(\text{IsParameterOf}) &\mapsto \text{isdataof}^{LD} \\
\alpha^{LD}(\text{IsMethodOf}) &\mapsto \text{ismethodof}^{LD} \\
\alpha^{LD}(\text{inheritsFrom}) &\mapsto \text{inheritsfrom}^{LD}
\end{aligned}
$$

$$
\begin{aligned}
\alpha^{LD}(\text{Accessess}) &\mapsto \text{uses}^{LD} \\
\alpha^{LD}(\text{Invokes}) &\mapsto \text{call}^{LD}
\end{aligned}
$$

Definition The LD value of a class $c \in class^{LD}$ is defined as:

$$
\begin{aligned}
S(c) &= succ^{+}(c, \text{inheritsfrom}^{LD}) \\
&\quad \textit{-- set of super classes c inherits from,} \\
&\quad \textit{-- excluding c itself} \\
V^{super}(c) &= \{v_i | v_i \in pred(S(c), \text{isdataof}^{LD} \wedge \\
&\quad v_i.visibility = protected)\} \\
&\quad \textit{-- set of inherited protected instance variables} \\
&\quad \textit{-- of super-classes of c} \\
V(c) &= \{v_i | v_i \in pred(C(c), \text{isdataof}^{LD} \wedge \\
&\quad v_i.visibility \neq public)\} \\
&\quad \textit{-- set of all non-public instance variables of c} \\
&\quad \textit{-- including static variables defined locally in M(c)} \\
V^{local}(c) &= V^{super}(c) \cup V(c) \\
&\quad \textit{-- set of all non-public class local} \\
&\quad \textit{-- instance variables of c} \\
M(c) &= pred(c, \text{ismethodof}^{LD}) \\
&\quad \textit{-- set of methods contained in c} \\
M^{*}(c) &= \{m_i | m_i.isGetterSetter \neq true \wedge m_i \in M(c)\} \\
&\quad \textit{-- set of methods contained in c,} \\
&\quad \textit{-- excluding trivial read/write methods} \\
M^{r/w}(c) &= \{m_i | m_i.isGetterSetter = true \wedge m_i \in M(c)\} \\
&\quad \textit{-- set of all read/write methods contained in c} \\
M^{r/w\ used}(c) &= \{m_i | m_i \in M^{r/w}(c) \cap succ(M^{*}(c), \text{call}^{LD})\} \\
&\quad \textit{-- set of all read/write methods contained in c and} \\
&\quad \textit{-- called by non-read/write methods contained in c} \\
T(c) &= \{v_i | v_i \in succ(M^{*}(c), \text{uses}^{LD})\} \\
&\quad \textit{-- set of all variables used in } M^{*}(c) \textit{ of c,} \\
&\quad \textit{-- excluding local variables defined in } M^{*}(c)
\end{aligned}
$$

$$
\begin{aligned}
L^{direct}(c) &= T(c) \cap V^{class-local}(c) \\
&\quad \textit{-- set of all variables of c directly used in } M^*(c), \\
&\quad \textit{-- excluding local variables, and variable used} \\
&\quad \textit{-- over getter/setter mtheods} \\
L^{indirect}(c) &= succ(M^{r/w-used}(c), \text{uses}^{LD}) \cap V^{class-local}(c) \\
&\quad \textit{-- set of all variables of c indirectly used in } M^*(c) \\
&\quad \textit{-- over read/write methods} \\
L(c) &= L^{direct}(c) \cup L^{indirect}(c) \\
LD(c) &= \frac{L(c)}{T(c)}
\end{aligned}
$$

Scale Ratio

Domain Rational in 0.0..1.0.

Highly Related Software Quality Properties

Re-Usability 2.4 is positively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability increases with increasing LD.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness increases with increasing LD.

Maintainability 2.6 decreases with increasing LD.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability increases with increasing LD.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability increases with increasing LD.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability increases with increasing LD.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability increases with increasing LD.

Portability 2.7 decreases with increasing LD.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability increases with increasing LD.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might decrease with increasing LD.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing LD.

Reliability 2.2 might decrease with increasing LD.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.

Fault-Tolerance might increase with increasing LD.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might increase with increasing LD.

Re-Usability 2.4 might decrease with increasing LD.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might increase with increasing LD.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might increase with increasing LD.

Efficiency 2.5 might decrease with increasing LD.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might increase with increasing LD.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might increase with increasing LD.

References

- LD is extensively discussed and evaluated in [11, 19, 2, 12],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Message Passing Coupling (MPC)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the MPC specific view).

Handle MPC

Description The MPC measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It allows for conclusions on the message passing (method calls) between objects of the involved classes. This allows for conclusions on re-useability, maintenance and testing effort.

Scope Class

View $V^{MPC} = (G^{MPC}, R^{MPC})$

- Grammar $G^{MPC} = (\{\text{class}^{MPC}\}, \emptyset, \text{class}^{MPC})$
- Relations $R^{MPC} : \{\text{call}^{MPC} : \text{class}^{MPC} \times \text{class}^{MPC}\}$
- Mapping α^{MPC}:

$$\begin{aligned} \alpha^{MPC}(\text{Class}) &\mapsto \text{class}^{MPC} \\ \alpha^{MPC}(\text{Invokes}) &\mapsto \text{call}^{MPC} \end{aligned}$$

Definition The MPC value of a class $c \in class^{MPC}$ is defined as:

$$MPC(c) = outdegree^*(c, \text{call}^{MPC})$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing MPC.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing MPC.

Maintainability 2.6 decreases with increasing MPC.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing MPC.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing MPC.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing MPC.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing MPC.

Portability 2.7 decreases with increasing MPC.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing MPC.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing MPC.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing MPC.

Reliability 2.2 might decrease with increasing MPC.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing MPC.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing MPC.

Re-Usability 2.4 might decrease with increasing MPC.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing MPC.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing MPC.

Efficiency 2.5 might decrease with increasing MPC.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since

they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing MPC.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing MPC.

References

- MPC is extensively discussed and validated in [14, 2, 4, 11, 19],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Package Data Abstraction Coupling ($PDAC$)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the DAC specific view).

Handle $PDAC$

Description The PDAC measures the coupling complexity caused by ADTs on package level. Based on DAC it transfers the effects of the coupling between classes on the reuse degree, maintenance and testing effort to more abstract organization units like packages and modules, which are as well decisively influenced by the coupling between classes of different packages.

Scope Package

View $V^{PDAC} = (G^{PDAC}, R^{PDAC})$

- Grammar $G^{PDAC} = (\{\text{class}^{PDAC}, \text{package}^{PDAC}\}, \emptyset,$ $\text{package}^{PDAC})$

- Relations
  R^{PDAC} : {referencestype^{PDAC} : class^{PDAC} $\times$ class^{PDAC},
  contains^{PDAC} : package^{PDAC} $\times$ class^{PDAC}}
- Mapping α^{PDAC}:

$$\begin{aligned} \alpha^{PDAC}(\text{Class}) &\mapsto \text{class}^{PDAC} \\ \alpha^{PDAC}(\text{Package}) &\mapsto \text{package}^{PDAC} \\ \alpha^{PDAC}(\text{IsOfType}) &\mapsto \text{referencestype}^{PDAC} \\ \alpha^{PDAC}(\text{Contains}) &\mapsto \text{contains}^{PDAC} \end{aligned}$$

Definition The $PDAC$ value of a package $p \in package^{PDAC}$ is defined as:

$$\begin{aligned} C(p) &= succ(p, \text{contains}^{PDAC}) \\ PDAC(p) &= \left| succ(C(p), \text{referencestype}^{PDAC}) \backslash C(p) \right| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing PDAC.

Attractiveness 2.4.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing PDAC.

Maintainability 2.6 decreases with increasing PDAC.

Analyzability 2.6.1: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing PDAC.

Changeability 2.6.2: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing PDAC.

Stability 2.6.3: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing PDAC.

Testability 2.6.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing PDAC.

Portability 2.7 decreases with increasing PDAC.

Adaptability 2.7.1: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing PDAC.

Related Software Quality Properties

Functionality 2.1 is both negatively and positively influenced by coupling.

Interoperability 2.1.3: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing PDAC.

Security 2.1.4: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.

Security might decrease with increasing PDAC.

Reliability 2.2 might decrease with increasing PDAC.

Fault-tolerance 2.2.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing PDAC.

Recoverability 2.2.3: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing PDAC.

Re-Usability 2.4 might decrease with increasing PDAC.

Learnability for Reuse 2.4.2: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing PDAC.

Operability for Reuse – Programmability 2.4.3: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing PDAC.

Efficiency 2.5 might decrease with increasing PDAC.

Time Behavior 2.5.1: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing PDAC.

Resource Utilization 2.5.2: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing PDAC.

References

- PDAC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

3.2.3 Cohesion

Lack of Cohesion in Methods ($LCOM$)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $LCOM$ specific view).

Handle $LCOM$

Description The Lack of Cohesion in Methods metric is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of method pairs not having instance variables in common, and the number of method pairs having common instance variables.

Scope Class

View $V^{LCOM} = (G^{LCOM}, R^{LCOM})$

- Grammar $G^{LCOM} = (\{\text{class}^{LCOM}, \text{method}^{LCOM}, \text{field}^{LCOM}\}, \emptyset, \text{class}^{LCOM})$
- Relations
  $R^{LCOM} : \{\text{methodof}^{LCOM} : \text{method}^{LCOM} \times \text{class}^{LCOM},$
  $\text{fieldof}^{LCOM} : \text{field}^{LCOM} \times \text{class}^{LCOM},$
  $\text{uses}^{LCOM} : \text{method}^{LCOM} \times \text{field}^{LCOM}\}$

- Mapping α^{LCOM}:

$$\begin{aligned}
\alpha^{LCOM}(\text{Class}) &\mapsto \text{class}^{LCOM} \\
\alpha^{LCOM}(\text{field}) &\mapsto \text{field}^{LCOM} \\
\alpha^{LCOM}(\text{Method}) &\mapsto \text{method}^{LCOM} \\
\alpha^{LCOM}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{LCOM} \\
\alpha^{LCOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{LCOM} \\
\alpha^{LCOM}(\text{Accessess}) &\mapsto \text{uses}^{LCOM}
\end{aligned}$$

Definition The $LCOM$ value of a class $c \in class^{LCOM}$, with n methods $m_1, m_2, \ldots, m_n \in M(c)$, having $\{I(c, m_i) \in F(c)\}$ represent the set of instance variables used by method m_i - there are n such sets $\{I_1, \ldots, I_n \in F(c)\}$ - is defined as:

$$\begin{aligned}
F(c) &= pred(c, \text{fieldof}^{LCOM}) \\
&\quad \textit{-- set of fields contained in } c \\
M(c) &= pred(c, \text{methodof}^{LCOM}) \\
&\quad \textit{-- set of methods contained in } c \\
I(c, m) &= succ(m, \text{uses}^{LCOM}) \cap F(c) \\
&\quad \textit{-- set of fields contained in } c \textit{ and used by } m \\
P(c) &= \{(m_i, m_j) | m_i, m_j \in M(c) \wedge I(c, m_i) \cap I(c, m_j) = \emptyset\} \\
&\quad \textit{-- set of disjunct sets of instant variables of } c \\
Q(c) &= \{(m_i, m_j) | m_i, m_j \in M(c) \wedge I(c, m_i) \cap I(c, m_j) \neq \emptyset\} \\
&\quad \textit{-- set of conjunct sets of instant variables of } c \\
LCOM(c) &= \begin{cases} |P(c)| - |Q(c)| & if\ |P(c)| > |Q(c)| \\ 0 & otherwise \end{cases}
\end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Reliability 2.2 is positively influenced by cohesion.

Maturity 2.2.1: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity decreases with increasing LCOM.

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability decreases with increasing LCOM.

Attractiveness 2.4.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness decreases with increasing LCOM.

Maintainability 2.6 decreases with increasing LCOM.

Analyzability 2.6.1: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability decreases with increasing LCOM.

Changeability 2.6.2: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.
Changeability decreases with increasing LCOM.

Stability 2.6.3: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability decreases with increasing LCOM.

Testability 2.6.4: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability decreases with increasing LCOM.

Portability 2.7 decreases with increasing LCOM.

Adaptability 2.7.1: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability decreases with increasing LCOM.

Related Software Quality Properties

Re-Usability 2.4 might decrease with increasing LCOM.

Learnability for Reuse 2.4.2: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing LCOM.

Operability for Reuse – Programmability 2.4.3: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing LCOM.

Efficiency 2.5 might decrease with increasing LCOM.

Time Behavior 2.5.1: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing LCOM.

Resource Utilization 2.5.2: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing LCOM.

Portability 2.7 decreases with increasing LCOM.

Replaceablity 2.7.4: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might decreases with increasing LCOM.

References

- LCOM is extensively discussed and evaluated in [5, 6, 14, 3, 2, 1, 7, 11, 12, 10, 4, 21, 17, 19],
- LCOM is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Improvement of LCOM ($ILCOM$)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $ILCOM$ specific view).

Handle $ILCOM$

Description The Improvement of LCOM (cf. Lack of Cohesion in Methods 3.2.3) metric is a measure for the number of connected components in a class. A component are methods of a class sharing (being connected by) instance variables of the class. The less separate components there are the higher is the cohesion of the methods in the class.

Scope Class

View $V^{ILCOM} = (G^{ILCOM}, R^{ILCOM})$

- Grammar $G^{ILCOM} = (\{\text{class}^{ILCOM}, \text{method}^{ILCOM}, \text{field}^{ILCOM}\}, \emptyset, \text{class}^{ILCOM})$
- Relations
  $R^{ILCOM} : \{\text{methodof}^{ILCOM} : \text{method}^{ILCOM} \times \text{class}^{ILCOM}, \text{fieldof}^{ILCOM} : \text{field}^{ILCOM} \times \text{class}^{ILCOM}, \text{uses}^{ILCOM} : \text{method}^{ILCOM} \times \text{fieILCOM}^{ILCOM}\}$
- Mapping α^{ILCOM}:

$$\begin{aligned} \alpha^{ILCOM}(\text{Class}) &\mapsto \text{class}^{ILCOM} \\ \alpha^{ILCOM}(\text{Field}) &\mapsto \text{field}^{ILCOM} \\ \alpha^{ILCOM}(\text{Method}) &\mapsto \text{method}^{ILCOM} \\ \alpha^{ILCOM}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{ILCOM} \\ \alpha^{ILCOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{ILCOM} \\ \alpha^{ILCOM}(\text{Accessess}) &\mapsto \text{uses}^{ILCOM} \end{aligned}$$

Definition The $ILCOM$ value of a class $c \in class^{ILCOM}$, with n methods $m_1, m_2, \ldots, m_n \in M(c)$, having $\{I(c, m_i) \in F(c)\}$ represent the

set of instance variables used by method m_i - there are n such sets $\{I_1, \ldots, I_n \in F(c)\}$ - is defined as:

$$\begin{aligned} F(c) &= pred(c, \text{fieldof}^{ILCOM}) \\ &\quad \textit{-- set of fields contained in } c \\ M(c) &= pred(c, \text{methodof}^{ILCOM}) \\ &\quad \textit{-- set of methods contained in } c \\ ILCOM(c) &= \left| scc(F(c) \cup M(c), \text{uses}^{ILCOM}) \right| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 1..\infty$.

Highly Related Software Quality Properties

Reliability 2.2 is positively influenced by cohesion.

Maturity 2.2.1: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity decreases with increasing ILCOM.

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability decreases with increasing ILCOM.

Attractiveness 2.4.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness decreases with increasing ILCOM.

Maintainability 2.6 decreases with increasing ILCOM.

Analyzability 2.6.1: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability decreases with increasing ILCOM.

Changeability 2.6.2: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.

Changeability decreases with increasing ILCOM.

Stability 2.6.3: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability decreases with increasing ILCOM.

Testability 2.6.4: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability decreases with increasing ILCOM.

Portability 2.7 decreases with increasing ILCOM.

Adaptability 2.7.1: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability decreases with increasing ILCOM.

Related Software Quality Properties

Re-Usability 2.4 might decrease with increasing ILCOM.

Learnability for Reuse 2.4.2: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing ILCOM.

Operability for Reuse – Programmability 2.4.3: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing ILCOM.

Efficiency 2.5 might decrease with increasing ILCOM.

Time Behavior 2.5.1: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing ILCOM.

Resource Utilization 2.5.2: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing ILCOM.

Portability 2.7 decreases with increasing ILCOM.

Replaceablity 2.7.4: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might decreases with increasing ILCOM.

References

- ILCOM is extensively discussed and evaluated in [11, 12, 2, 19],
- ILCOM is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Tight Class Cohesion (TCC)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the TCC specific view).

Handle TCC

Description The Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a low cohesion indicate errors in the design.

Scope Class

View $V^{TCC} = (G^{TCC}, R^{TCC})$

- Grammar $G^{TCC} = (\{\text{class}^{TCC}, \text{method}^{TCC}, \text{field}^{TCC}\}, \emptyset, \text{class}^{TCC})$
- Relations $R^{TCC} : \{\text{methodof}^{TCC} : \text{method}^{TCC} \times \text{class}^{TCC}, \text{fieldof}^{TCC} : \text{field}^{TCC} \times \text{class}^{TCC}, \text{uses}^{TCC} : \text{method}^{TCC} \times \text{field}^{TCC}\}$

- Mapping α^{TCC}:

$$\begin{aligned}
\alpha^{TCC}(\text{Class}) &\mapsto \text{class}^{TCC} \\
\alpha^{TCC}(\text{Field}) &\mapsto \text{field}^{TCC} \\
\alpha^{TCC}(\text{Method}) &\mapsto \text{method}^{TCC} \\
\alpha^{TCC}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{TCC} \\
\alpha^{TCC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{TCC} \\
\alpha^{TCC}(\text{Accessess}) &\mapsto \text{uses}^{TCC}
\end{aligned}$$

Definition The Tight Class Cohesion (TCC) measures the ratio between the actual number of visible directly connected methods in a class $NDC(C)$ divided by the number of maximal possible number of connections between the visible methods of a class $NP(C)$. Two visible methods are directly connected, if they are accessing the same instance variables of the class. n is the number of visible methods leading to:

$$\begin{aligned}
M(c) &= pred(c, \text{methodof}^{TCC}) \\
&\quad \textit{-- set of methods contained in } c \\
M^{public}(c) &= \{m_i | m_i.\text{visibility} = \text{public} \wedge m_i \in M(c)\} \\
&\quad \textit{-- set of methods contained in } c \\
&\quad \textit{-- having public visibility} \\
I(c,m) &= succ(m, \text{uses}^{TCC}) \cap F(c) \\
&\quad \textit{-- set of fields contained in } c \textit{ and used by } m \\
NDC(c) &= \{(m_i, m_j) | I(c,m_i) \cap I(c,m_j) \neq \emptyset \wedge m_i, m_j \in M(c)\} \\
&\quad \textit{-- set of public method pairs contained in } c \\
&\quad \textit{-- accessing the same field} \\
NP(c) &= \frac{\left|M^{public}(c)\right| \left(\left|M^{public}(c)\right| - 1\right)}{2} \\
TCC(c) &= \frac{NDC(c)}{NP(c)}
\end{aligned}$$

Scale Ratio

Domain Rational $\in$ 0.0..1.0.

Highly Related Software Quality Properties

Reliability 2.2 is positively influenced by cohesion.

Maturity 2.2.1: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity increases with increasing TCC.

Re-Usability 2.4 is negatively influenced by coupling.

Understandability for Reuse 2.4.1: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability increases with increasing TCC.

Attractiveness 2.4.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness increases with increasing TCC.

Maintainability 2.6 increases with increasing TCC.

Analyzability 2.6.1: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability increases with increasing TCC.

Changeability 2.6.2: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.
Changeability increases with increasing TCC.

Stability 2.6.3: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability increases with increasing TCC.

Testability 2.6.4: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability increases with increasing TCC.

Portability 2.7 increases with increasing TCC.

Adaptability 2.7.1: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability increases with increasing TCC.

Related Software Quality Properties

Re-Usability 2.4 might decrease with increasing TCC.

Learnability for Reuse 2.4.2: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing TCC.

Operability for Reuse – Programmability 2.4.3: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing TCC.

Efficiency 2.5 might decrease with increasing TCC.

Time Behavior 2.5.1: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing TCC.

Resource Utilization 2.5.2: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing TCC.

Portability 2.7 increases with increasing TCC.

Replaceablity 2.7.4: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might increases with increasing TCC.

References

- TCC is extensively discussed and evaluated in [3, 2, 19],
- TCC is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

3.3 Design Guidelines and Code Conventions

3.3.1 Documentation

Lack Of Documentation (LOD)

Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end.

Handle LOD

Description How many comments are lacking in a class, considering one class comment and a comment per method as optimum. Structure and content of the comments are ignored.

Scope Class

View $V^{LOD} = (G^{LOD}, R^{LOD})$

- Grammar $G^{LOD} = (\{\text{scope}^{LOC}\}, \emptyset, \text{scope}^{LOC})$
- Relations $R^{LOD} : IsMethodOf$
- Mapping α^{LOD}:

$$\begin{aligned} \alpha^{LOD}(\text{Class}) &\mapsto \text{scope}^{LOD} \\ \alpha^{LOD}(\text{Method}) &\mapsto \text{method}^{LOD} \\ \alpha^{LOD}(\text{IsMethodOf}) &\mapsto \text{containsMethod}^{LOC} \end{aligned}$$

Definition The LOD value of a element $c \in scope^{LOD}$ is defined as:

$$\begin{aligned} T(c) &= succ(c, containsMethod) \cup c \\ D(c) &= \{e \in T(c) | e.hasJavaDoc == true\} \\ LOD(c) &= 1.0 - \frac{|D(c)|}{|T(c)|} \end{aligned}$$

Scale Rational.

Domain Rational numbers $\in 0.0..1.0\infty$.

Highly Related Software Quality Properties

Re-Usability 2.4 is inversely influenced by LOD.

Understandability for Reuse 2.4.1: Understanding if a class is suitable for reuse depends on its degree of documentation. Understandability decreases with increasing LOD.

Learnability for Reuse 2.4.2: Learning if a class is suitable for reuse depends on the degree of documentation.
Learnability decreases with increasing LOD.

Operability for Reuse – Programmability 2.4.3: How well a class can be integrated depends on the degree of documentation.
Programmability decreases with increasing LOD.

Maintainability 2.6 decreases with increasing LOD.

Analyzability 2.6.1: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its degree of documentation.
Analyzability decreases with increasing LOD.

Changeability 2.6.2: Changing a class requires prior understanding, which, in turn, is more complicated for undocumented classes.
Changeability decreases with increasing LOD.

Testability 2.6.4: Writing test cases for classes and methods requires understanding, which, in turn, is more complicated for undocumented classes.
Testability decreases with increasing LOD.

Portability 2.7 decreases with increasing LOD.

Adaptability 2.7.1: As for changeability 2.6.2, the degree of documentation of software has a direct impact. Each modification requires understanding which is more complicated for undocumented systems.
Adaptability decreases with increasing LOD.

Replaceablity 2.7.4: The substitute of a component must imitate its interface. Undocumented interfaces are difficult to check for substitutability and to actually substitute.
Replaceablity decline with increasing LOD.

Related Software Quality Properties

Reliability 2.2 decreases with increasing LOD.

Maturity 2.2.1: Due to reduced analyzability 2.6.1 and testability 2.6.4, bugs might be left in undocumented software. Therefore, maturity may be influenced by degree of documentation.
Maturity decreases with increasing LOD.

Re-Usability 2.4 is inversely influenced by LOD.

Attractiveness 2.3.4: Attractiveness of a class depends on its adherence to coding conventions such as degree of documentation.
Attractiveness decreases with increasing LOD.

Maintainability 2.6 decreases with increasing LOD.

Stability 2.6.3: Due to reduced analyzability 2.6.1 and testability 2.6.4, also stability may be influenced negatively by size.
Stability decreases with increasing LOD.

References

- LOD is implemented in the VizzAnalyzer Metric Suite.

3.3.2 Code Conventions

None so far.

Bibliography

[1] N. V. Balasubramanian. Object-Oriented Metrics. In *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[2] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook. `http://www.iam.unibe.ch/~famoos/handbook/`, October 1999.

[3] J. M. Bieman and B. Kang. Cohesion and Reuse in an Object-Oriented System. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 259–262, New York, NY, USA, 1995. ACM Press.

[4] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999.

[5] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM Press.

[6] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering*, volume 20 (6), pages 476–493, June 1994.

[7] N. Churcher and M. Shepperd. Comments on "A Metrics Suite for Object Oriented Design". In *IEEE Transactions on Software Engineering*, volume 21 (3), pages 263–265, 1995.

[8] F. Brito e Abreu and R. Carapua. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *4th International Conference on Software Quality*, October 1994. McLean, Virginia, USA.

[9] R. Harrison, S. J. Counsell, and R. V. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Trans. Softw. Eng.*, 24(6):491–496, 1998.

[10] 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.

[11] M. Hitz and B. Montazeri. Measure Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95)*, October 1995.

[12] M. Hitz and B. Montazeri. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Trans. Softw. Eng.*, 22(4):267–271, 1996.

[13] W. Humphrey. *Introduction to the Personal Software Process*. SEI Series in Software Engineering. Addison-Wesley, 1997.

[14] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. *Software Metrics Symposium, 1993. Proceedings., First International*, pages 52–60, 1993.

[15] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proc. of the 1st Int. Software Metrics Symposium*, pages 52–60, May 1993.

[16] R. Martin. OO Design Quality Metrics – An Analysis of Dependencies (position paper). In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, oct 1994.

[17] T. Mayer and T. Hall. A Critical Analysis of Current OO Design Metrics. *Software Quality Control*, 8(2):97–110, 1999.

[18] Th. J. McCabe and Ch. W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[19] Th. Panas, R. Lincke, J. Lundberg, and W. Löwe. A Qualitative Evaluation of a Software Development and Re-Engineering Project. In *SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, pages 66–75, Washington, DC, USA, 2005. IEEE Computer Society.

[20] D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An Extensible Meta-Model for Program Analysis. In *22nd IEEE International Conference on Software Maintenance*, 2006.

[21] R. Subramanyam and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[22] A. H. Watson and Th. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication 500-235*, 1996.