Georg-Simon-Ohm-Hochschule Nürnberg
Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Masterarbeit
von

Dipl.-Inform. (FH) Thomas Görtz

Wintersemester 2009

Metriken im Kontext modellgetriebener Software-Entwicklung

Erstprüfer: Prof. Dr. Hans-Georg Hopf
Zweitprüfer: Prof. Dr. Jürgen Wohlrab
Betreuer: Dipl. Ing. (FH) Peter Schedl

Thomas Görtz
Eppinghovener Straße 23
46535 Dinslaken

Ich bestätige, dass ich die Abschlussarbeit mit dem Titel:

Metriken im Kontext modellgetriebener Software-Entwicklung

gemäß § 22 (7) APO selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Dinslaken, den 10. Dezember 2009

(Unterschrift)

_____

Thomas Görtz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Zielsetzung der Arbeit

IBM Rational bietet mit Rhaposdy ein populäres Werkzeug im Bereich der modellgetriebenen Software-Entwicklung für den Embedded-Bereich auf Basis der UML 2 an. Das Ziel der Arbeit ist der Entwurf und die Implementierung einer Applikation zur Bestimmung der Qualität von Modellen, welche mit Rhapsody erstellt werden. Zu diesem Zweck wird eine Auswahl der dazu notwendigen Mess- und Bewertungsverfahren getroffen. Die Umsetzung soll im Rahmen einer experimentellen Messung beurteilt werden.

1.2 Aufbau und Kapitelübersicht

Das Vorgehen der Arbeit gliedert sich in die folgend beschriebenen Kapitel:

Kapitel 2, *Grundlagen*, führt die für die Auswahl der Verfahren nötigen Grundlagen zur modellgetriebenen Software-Entwicklung, Metriken, Qualitätsmodellen und Messtheorie ein.

Kapitel 3, *Umsetzung*, zeigt den Auswahlprozess der für die Anwendung geeigneten Messmethoden. Ergänzend werden die dazu gehörenden Metriken definiert und beschrieben.

Kapitel 4, *Softwarearchitektur*, stellt die Struktur und das Verhalten der zu erstellenden Messanwendung dar.

Kapitel 5, *Experimentelle Messung*, stellt die Messergebnisse und deren Interpretation der umgesetzten Lösung anhand von Rhapsody-Projekten dar.

Kapitel 6, *Schlussbetrachtungen*, betrachtet abschließend die Ergebnisse der Arbeit und bietet einen Ausblick auf ein weiteres Vorgehen. [1]

[1] Der vorliegende Text ist auf Basis des Latex-Templates von Tilo Gockel [Goc08] erstellt.

Kapitel 2

Grundlagen

2.1 Einleitung

Zum vierzigsten Jubiläum der NATO-Konferenz zum Thema Software-Entwicklung spricht Tom DeMarco vom Ende dieser noch jungen Disziplin [DeM09]. Derselbe Tom DeMarco prägte diesbezüglich die berühmte Aussage, was man nicht messen könne, könne man nicht kontrollieren. Diese Kontroverse macht es gerade heute notwendig, eine der vermeintlichen *Silver Bullets* der neunziger Jahre mit neuen Augen im Kontext heutiger Herausforderungen zu betrachten: die Software-Metriken.

Neben der Entwicklung, die Kommunikation in den Entwicklungsprozessen immer mehr in den Mittelpunkt zu bringen, brachten die vergangenen Jahre auch viele Neuerungen im Bereich der Technologie. Zunehmende Komplexität der Domänen und Anforderungen führten zu dem Wunsch nach mehr Abstraktion der Mittel. Hierzu gehört auch die modellgetriebene Software-Entwicklung, welche immer mehr an Bedeutung gewinnt.

In den Anfängen wurde Modellierung fast ausschließlich zur Strukturierung und Dokumentation im Entwicklungsprozess genutzt. Heutzutage ist die Modellierung bis hinunter zum Verhalten der Klassen und deren Codegenerierung alltäglich. Mit etwas Verzug hält diese Entwicklung auch im Bereich der eingebetteten Anwendungen Einzug. Die Frage der Entwickler nach objektiven Bewertungsmöglichkeiten bei der Umsetzung bleibt aber dieselbe wie zu Beginn der Software-Entwicklung. Die Schwierigkeit beim Auffinden von Antworten leider auch.

Eine Methode zur Bewertung von Qualität ist die Messung des Produkts und die Beurteilung der Ergebnisse gegen validierte Qualitätsmodelle. In diesem Kapitel sollen die Grundlagen eingeführt werden, um diese Methode auf das Werkzeug Rhapsody anwenden zu können. Begonnen wird mit der Definition der Sachdomäne der modellgetriebenen Software-Entwicklung.

2.2 Modellgetriebene Softwareentwicklung

2.2.1 Einleitung

In der Software-Entwicklung ist die Verwendung von Modellen schon seit Langem üblich. Bei dieser Methodik handelt es sich aber meist nur um einen „modellbasierten“ Ansatz der Modellnutzung. Dabei sind Modell und Implementierung getrennt, wobei das Modell hier der Dokumentation dient. Diese Trennung erfordert eine hohe Disziplin der Entwickler, um Modell und Code während der Entwicklung nicht auseinander laufen zu lassen. Die Umsetzung des Modells hin zum Quellcode muss zudem vom Entwickler geleistet werden und ist eine sichere Quelle für Fehler. Im Kontext der modellgetriebenen Software-Entwicklung hingegen ist handgeschriebener Code mit dem Modell gleichzusetzen.

Im folgenden Kapitel werden die wichtigsten Begriffe dieser Methodik definiert. Des Weiteren wird eine Eingliederung des Ansatzes im Kontext eines Software-Lebenszyklusses vorgenommen. Eine Einführung der Problemdomäne schließt das Kapitel ab.

2.2.2 Definition der Begriffe

Die in der Einleitung beschriebene Gleichsetzung von Code und Modell in der modellgetriebenen Softwareentwicklung wird wie folgt definiert:

> „Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ [SVH07]

Die Umsetzung eines Modells in ausführbaren Code erfolgt automatisiert und muss nicht mehr vom Entwickler durchgeführt werden. Dabei ist eine möglichst zugängliche Abstraktion der Domäne zu suchen. Diese findet üblicherweise ihren Ausdruck in visueller Form. Komplexe Zusammenhänge können mit der Modellierung durch UML-Diagramme, BPML-Diagramme, Petri-Netze, ereignisgesteuerte Prozessketten, usw. für den Nutzer zugänglich [PT07] dargestellt werden. Aber auch die textuelle Modellierung ist möglich.

Um die grundlegenden Begriffe der MDSD zu definieren, werden diese in Abbildung 2.1 in Bezug zueinander dargestellt [SVH07].

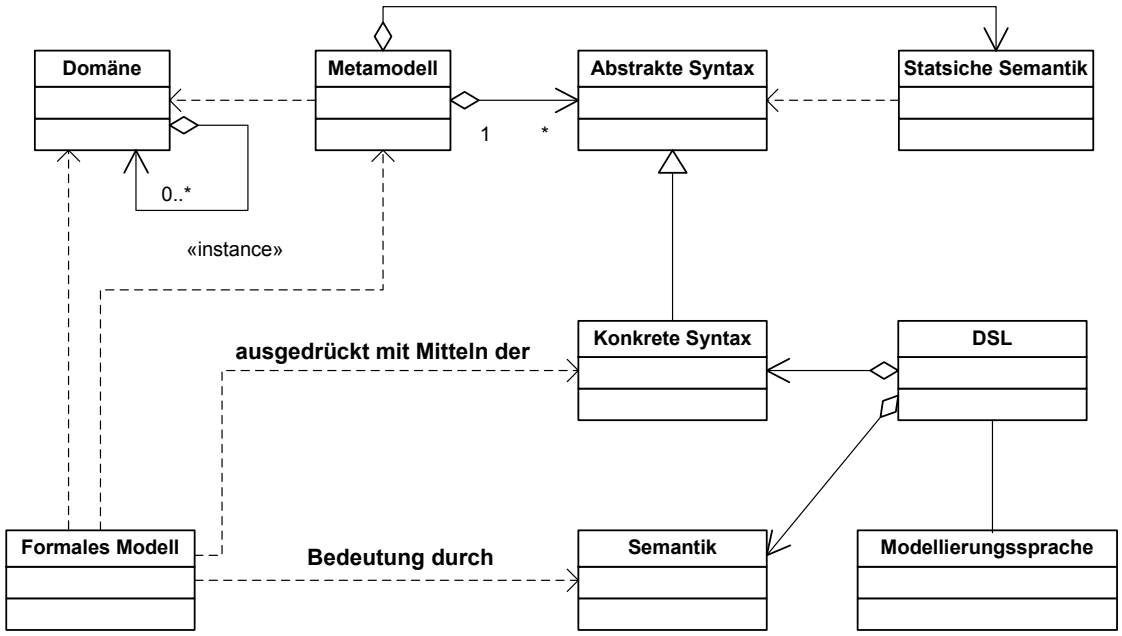


Abbildung 2.1: Definition der Begriffe im Kontext der MDSD.

Domäne

In [SVH07] wird die Domäne als der Arbeitskontext beschrieben. Grundsätzlich kann hier die Unterscheidung von der fachlichen zur architekturzentrierten Domäne getroffen werden. Erstere ist die Beschreibung für den Problemkontext, wie z.B. die „Flugraumüberwachung“ und deren Konzepte wie Luftstraßen, Streckennetz, Luftraumbelastung etc. Die architekturspezifische Domäne ist dagegen technischer Natur und beschreibt die Aspekte der Softwarearchitektur, wie Komponenten, Klassen, etc.

Metamodell

Abbildung 2.2 zeigt die Beschreibung der realen Welt durch ein Modell. Die Meta-Modellierung wiederum beschreibt die Eigenschaften des Modells [PT07]. Das Modell ist als eine Instanz des Metamodells zu verstehen.

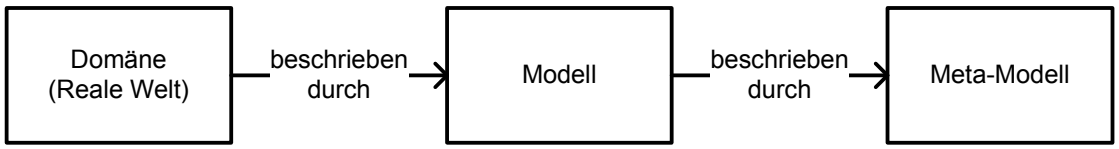


Abbildung 2.2: Zusammenhänge der Metamodellierung.

Eines der bekanntesten Metamodelle ist z.B. die Meta Object Facility (MOF) der Object Management Group. Genauer betrachtet ist dies sogar ein Meta-Metamodell, aus dem sich das Metamodell der UML instantiiert. Daraus können wiederum konkrete UML-Modelle instantiiert werden. Grundsätzlich formalisiert es die Struktur einer Domäne und umfasst die abstrakte und statische Semantik einer Sprache [SVH07].

Syntax und statische Semantik

Die abstrakte Syntax eines Metamodells ist die Definition der Metamodellelemente ohne Vorgabe der Beschreibungsform. Sie beschreibt, welche Elemente das Metamodell besitzt, sowie deren Beziehung zueinander. Deren Modellierungsregeln werden von der statischen Semantik definiert. Diese sogenannten Constraints des Metamodells geben vor, welche Randbedingungen vom Modell eingehalten werden müssen, damit es wohldefiniert und gültig ist. Das Modell selbst wird in der abgeleiteten konkreten Syntax beschrieben. Diese ist in der Form unabhängig von der abstrakten Syntax und kann visueller oder textueller Natur sein.

Domänenspezifische Sprache

Die domänenspezifische Sprache (DSL) besteht aus Metamodell, statischer Semantik und deren konkreter Syntax. Diese Sprache ist als Programmiersprache für eine Domäne anzusehen [SVH07] und erhält deren Bedeutung durch die formale Definition einer dynamischen Semantik. Ihre Verwirklichung erhält diese Sprache meist mit der Umsetzung in eine Programmiersprache mittels zugehörigem Generator. Wie die Domäne kann auch die DSL in fachliche und architekturspezifische Sprache unterteilt werden.

Formales Modell

Um mit Hilfe der DSL ausführbaren Code erzeugen zu können, muss ein formales Modell als Programm modelliert werden. Dieses Modell soll durch Generierung oder Interpretierung in lauffähigen Code überführbar sein. Es ist üblich, dass formale Modelle mit manuell erzeugten Quelltexten angereichert werden, um diese für das Zielsystem vollständig zu definieren.

2.2.3 Domäne

Mit Rhapsody ist die in dieser Arbeit betrachtete Domäne architekturspezifisch (s.h. Kapitel 2.2.2). Das Werkzeug richtet sich an den Einsatz in der Entwicklungsdomäne der Embedded-Anwendungen (s. Abbildung 2.3) [LC08].

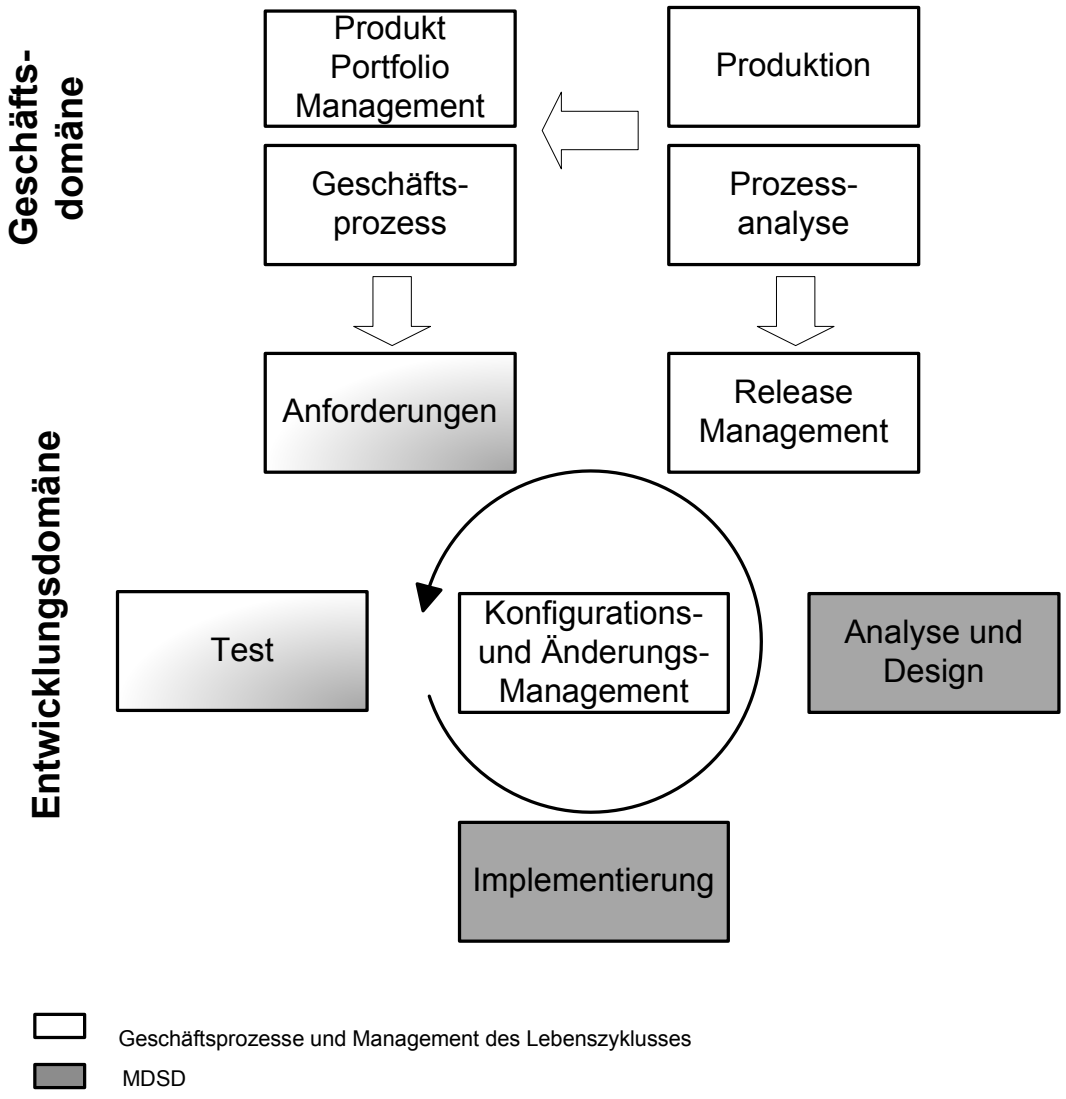


Abbildung 2.3: Architekturspezifische Domäne des Werkzeugs Rhapsody.

Die Kernfunktion von Rhapsody ist die visuelle Modellierung und Code-Generierung auf Basis von UML 2 Modellen für verschiedene Plattformen. Auf System- und Architekturniveau wird zudem die Modellierung mittels SysML und verschiedener Architektur-Standards wie AUTOSAR, MODAF oder DoDAF als UML-Profile unterstützt. Ergänzt wird Rhapsody durch eine Werkzeugsammlung, welche Funktionen wie Report-Generierung, automatische Testgenerierung, Anforderungsverfolgung oder die eigene Definition von Code-Generatoren liefert.

Die für diese Arbeit zu betrachtenden Modellmerkmale sollen die der objektorientierten Analyse, des Designs und der Implementierung architekturspezifischer Konzepte sein. Dafür ist es wichtig, auf die ereignisgetriebene Architektur als weiteres Merkmal von Rhapsody einzugehen. Damit der aus dem Modell generierte Code auf dem Zielsystem integriert werden kann, wird ein Laufzeitrahmen (Middleware) benötigt. Dieser Rahmen implementiert die ereignisgetriebene Kommunikation zwischen den modellierten Klassen und soll in die messtechnische Betrachtung einbezogen werden.

2.2.4 Verteilung der Artefaktmassen

Wie in Kapitel 2.2.2 beschrieben, kommt auch die MDSD in der Regel nicht ohne handgeschriebenen Code aus, um das Modell endgültig für die Generierung zu formalisieren. Der Anteil der handgeschriebenen, nicht modellierten Code-Masse reduziert sich dabei auf 34 Prozent [SVH07]. Die Entwicklungstendenz der Verteilung der Code-Massen ist in Abbildung 2.4 für die MDSD im Embedded-Bereich dargestellt [LC08].

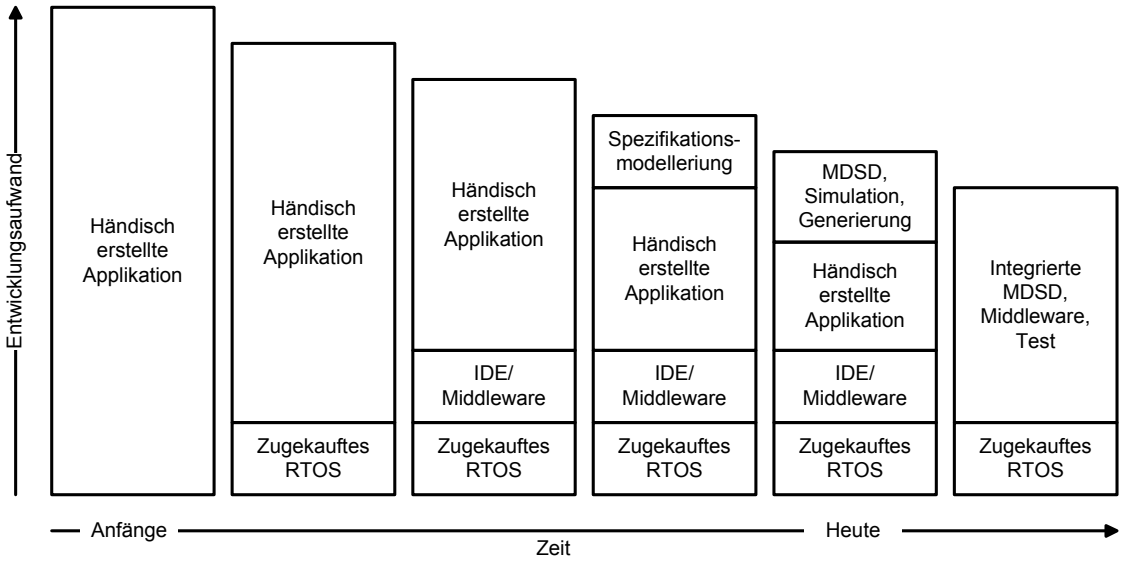


Abbildung 2.4: Tendenz der MDSD-Artefaktmassen im Embedded Bereich.

Bei der Umsetzung einer geeigneten Messanwendung muss dieser Umstand berücksichtigt werden. Bei der Auswahl der dafür sinnvollen Metriken sollte daher die reine Betrachtung des Modells, unter Ausschluss der manuell erstellten Artefaktmasse, vermieden werden.

2.3 Metriken

2.3.1 Einleitung

Abbildung 2.5: Karikatur: „Was zum Teufel pro Minute“-Metrik.

Natürlich gibt es nicht nur die eine wahre Metrik für die Software-Messung, wie in Abbildung 2.5 karikiert [Mar08]. Allerdings zeigt diese überspitzte Darstellung schon wichtige Fragestellungen, welche bei der Klassifikation von Metriken entstehen. Dabei können Merkmale wie Subjektivität, Korrelationen, Auswahl des Messgegenstands und viele mehr eine Rolle spielen. Daher existieren zahlreiche Definitionen und Klassifikationen [SB99], die alle ihre Berechtigung haben:

- Produkt- und Prozessmetriken
- Design- und Projektmetriken
- Objektive und subjektive Metriken
- Direkte und indirekte Metriken
- Explizite und abgeleitete Metriken
- Absolute und relative Metriken
- Dynamische und statische Metriken
- Vorhersagende und erklärende Metriken

In dieser Arbeit soll es in erster Linie um direkte objektive sowie indirekte subjektive Produktmetriken gehen. Bei Ersteren handelt es sich um die messtechnische Quantifizierung ausgewählter Attribute eines fertigen oder sich noch in der Entstehung befindlichen Software-Entwicklungsprodukts [SB99]. Bei dem Messgegenstand handelt es sich um die Ergebnisse der Entwicklungsaktivitäten der objektorientierten MDSD (s. Kapitel 2.2). Sie beziehen sich nicht auf die Messung anderer Charakteristika und stehen für sich allein.

Interpretiert man aus Korrelation dieser Metriken weitere Eigenschaften des Messgegenstandes, spricht man von indirekten und subjektiven Metriken. Das Bilden von Zusammenhängen involviert das menschliche Urteilsvermögen und dessen Erfahrung.

Zur Verdeutlichung des Einsatzgebiets folgt die Eingliederung in das Qualitätsmanagement und dessen Maßnahmen.

2.3.2 Einordnung in das Qualitätsmanagement

Total Quality Management

Stellvertretend ist in Abbildung 2.6 anhand des *Total Quality Management* der Einsatz von Metriken im Qualitätsmanagement erkennbar. Metriken sind hier im Kontext von Qualitätsmodellen (s. Kapitel QModelle) Fundament und kontinuierlicher Prozessbegleiter [Kan03].

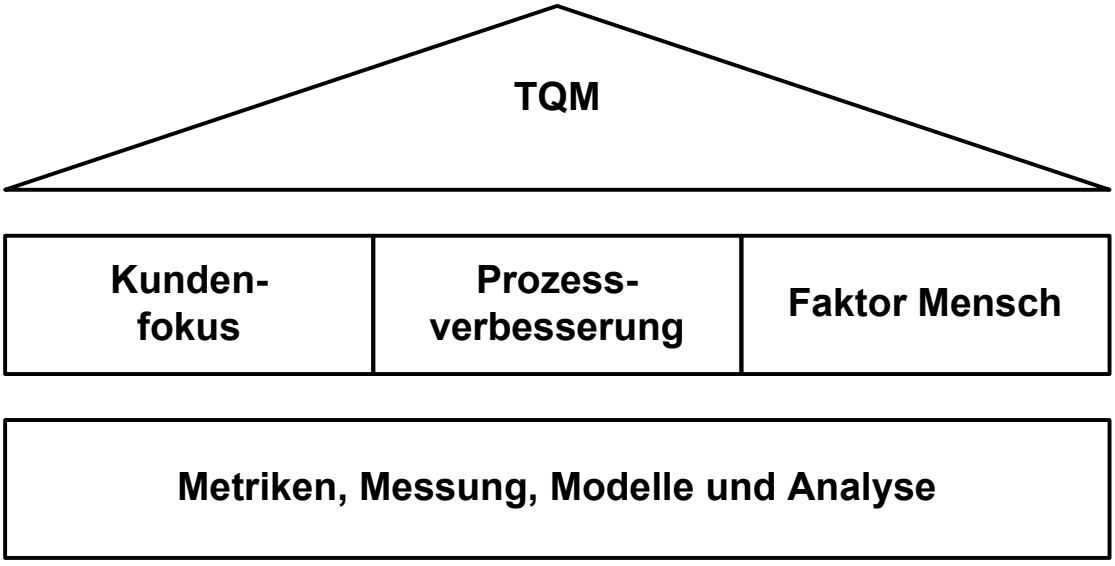


Abbildung 2.6: Übersicht Total Quality Management.

QM-Maßnahmen

Das gilt auch für die Prozessverbesserung im Sinne der Produktentwicklung. Hier können aus anderer Sicht die Metriken in verschiedene Bereiche der QM-Maßnahmen eingegliedert werden (s. Abbildung 2.7) [Hop08] [Bal98].

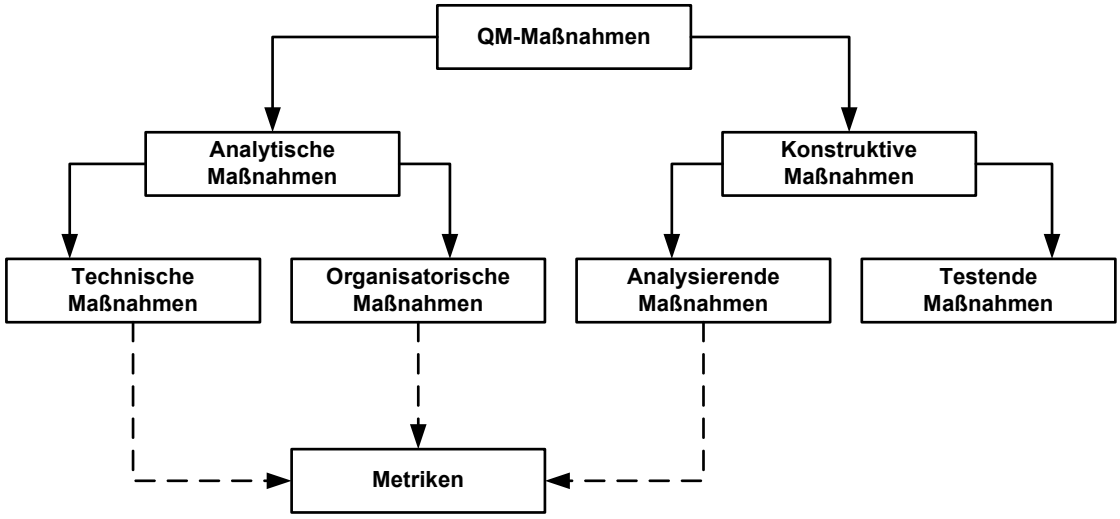


Abbildung 2.7: Metriken aus Sicht der QM-Maßnahmen.

Generell ist für den Einsatz von Metriken die Unterstützung durch das Management und die Eingliederung in den Qualitätsprozess notwendig [AKP02]. Die reine Einführung einer Messmethode und ihrer Werkzeuge ohne die notwendige Prozessreife anzustreben erzeugt ansonsten nur einen geringen Nutzwert der Ergebnisse und unnötigen Messaufwand [Hop08]. Im schlimmsten Fall kann der falsche Einsatz von Messmethoden und deren Fehlinterpretationen zu einer qualitativ negativen Steuerung von Entwicklungsprozessen führen. Besonders sollten bei der Einführung von Messmethoden soziale Aspekte mit bedacht werden. Die Bewertung von Arbeitsergebnissen ist auch immer ein intimer Eingriff in die persönlichen Arbeitsbereiche der Beteiligten. Unachtsamer Umgang kann hier zu mehr Schaden als Nutzen führen.

2.3.3 Geschichte der objektorientierten Metriken

Der in dieser Arbeit zentrale Bereich von Produktmetriken im Bezug auf objektorientierte Software-Anwendungen hat seinen Anfang in den 50'er Jahren mit dem Zählen von Programmzeilen. Anhand der Tabelle 2.1 ist ersichtlich, dass die Blütezeit der Produktmetriken in den 90'ern zu finden ist [Wan09]. In dieser Phase wurden die Erfahrungen und Metriken der Pioniere an das neue Paradigma der Objektorientierung angepasst. Damit ging auch die zunehmende Integration der Metriken in Qualitätsmodellen und deren Validierung einher. Diese sollen im folgenden Kapitel betrachtet werden.

Zeitpunkt	Werk
1950er	LOC (Lines of Code). Es wird vermutet, Grace Hopper setzte diese Metrik ein, um die Größe des ersten Compilers A-0 zu messen (1952)
1974	Erste ernsthafte Versuche der Produktivitätsbestimmung mithilfe der LOC-Metrik von Wolverton [Wol74].
1976	Einführung der McCabe-Metriken [McC76].
1977	Erstes Buch über Software-Metriken von Tom Gilb [Gil77].
1977	Einführung der Halstead-Metriken [Hal77].
1978	Boehm schreibt Buch mit Richtlinien zum Einsatz von Metriken [BBK+78].
1979	Einführung der Function Points Metrik [Alb79].
1988	Rocacher veröffentlicht erste Arbeit zu OO-Metriken im Kontext von Smalltalk [Roc88].
1989	Morris veröffentlicht 9 OO-Metriken [Mor89].
1989	Lieberherr/Holland präsentieren „Law of Demeter“ [LH89].
1994	Metrik-Suite von Chidamber/Kemerer mit 6 Metriken [SRK94]. Qualitätsmodell **MOOSE**.
1994	Erstes Buch über OO-Metriken von Lorenz/Kidd [LK94]. Qualitätsmodell **LK OO Metriken**
1995	Metrics for Object-Oriented Design suite **(MOOD)** von Brito/Carpuca [Abr95].
1996	Buch über OO-Metriken von Henderson-Sellers [HS96].
2002	Qualitätsmodell **QMOOD** von Bansiya/Davis [BD02].
2006	Praxisorientierte Anwendung von OO-Metriken von Lanza/Marinescu [LM06]. Qualitätsmodell **OOMIP**.

Tabelle 2.1: Kurze Geschichte der OO-Produktmetriken.

2.4 Qualitätsmodelle

2.4.1 Einleitung

Selbst die elegantesten Metriken erzeugen nur einen Haufen Zahlen, sieht man diese nicht im Kontext eines geeigneten Qualitätsmodells.

Dieses Kapitel stellt die Goal-Question-Metric-Methode vor, welche einen Weg zu Entwurf, Anwendung und Validierung eines auf Messung beruhenden Qualitätsmodells zeigt. Um eine Aussage über die Qualität des Modells selbst treffen zu können, werden Bewertungskriterien benötigt. Bei Betrachtung oder Auswahl objektorientierter Qualitätsmodelle werden in diesem Zusammenhang die erwünschten Eigenschaften von qualitativen Referenzmodellen beschrieben [EWEBBF04]. Es folgt eine Betrachtung und der Vergleich gängiger objektorientierter Qualitätsmodelle. Diese sollen in der Umsetzung zur Auswahl stehen, um einen passenden Ansatz zur Qualitätsbestimmung, in Bezug auf MDSD, finden zu können.

2.4.2 Die GQM-Methode

Mithilfe der GQM-Methode können auf Messung basierende Qualitätsmodelle entwickelt werden. Das Vorgehen schlägt dafür vier Phasen vor (s. Abbildung 2.8) [SB99].

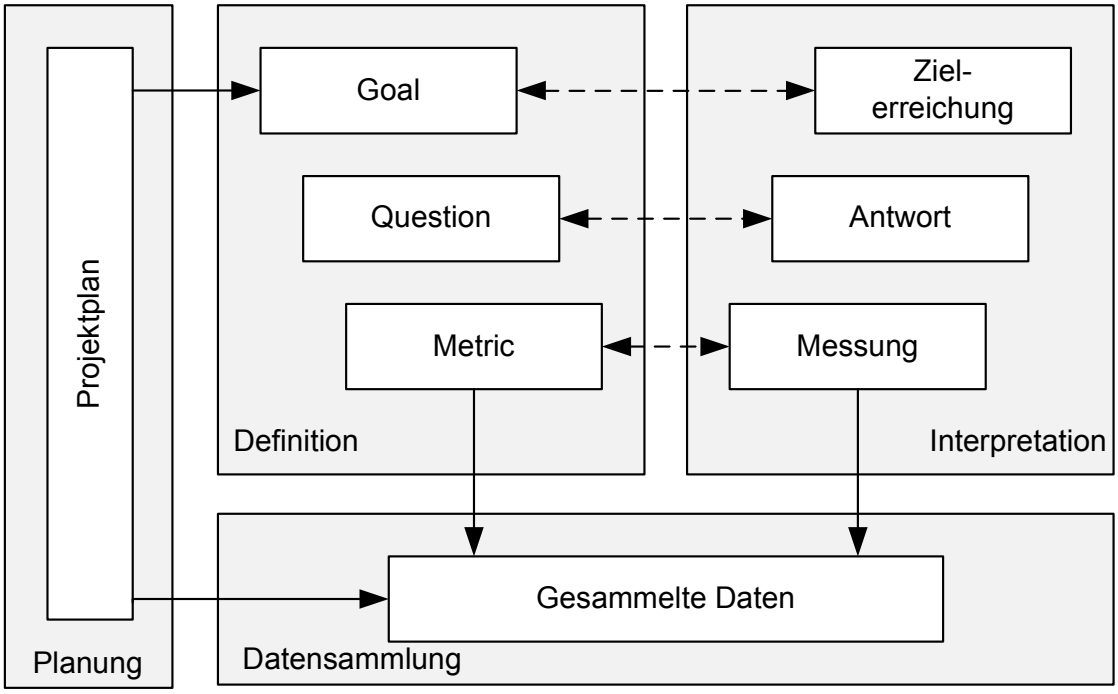


Abbildung 2.8: Das GQM-Modell.

Die Planungsphase konzentriert sich auf das klassische Projektmanagement des Messvorhabens. Hier wird das Projekt initiiert, die Beteiligten zusammengestellt, die zeitliche Planung festgehalten, Schulungsmaßnahmen organisiert etc. Die GQM-Methode trägt die zentrale und folgende Phase seines Konzepts bereits im Namen. In dieser wird das zu erreichende Qualitätsziel definiert. Für dieses Ziel werden Fragen gestellt, welche zum Erreichen des Ziels beantwortet werden sollen. Die Antwort auf jede Frage wird dann wiederum durch passende Metriken gefunden. Wichtiger Bestandteil dieser Phase ist das Aufstellen einer Hypothese, welche die Erwartungen an die Messungen zu späterer Validierung formuliert. In der Messphase wird die für die Messung notwendige Datenbasis etabliert. Während der abschließenden Interpretationsphase werden die Messergebnisse der Datensammlung gedeutet, um die vorher gestellten Fragen zu beantworten. Diese Antworten sollen klären, ob das gestellte Ziel erreicht wurde [SB99] [AKP02].

Der zentrale Ansatz bei der GQM-Methode ist der, dass das Messen von Prozessen und Produkten zielorientiert durchgeführt wird. Mit GQM können gewünschte Ziele systematisch in bestehende Qualitätsmodelle und Prozesse integriert werden. Natürlich müssen deshalb nicht gleich immer ganze Organisationen zielorientiert gemessen und optimiert werden. Der GQM-Ansatz hilft generell, das zielorientierte Messen auch mit kleineren Zielen methodisch durchzuführen.

An das entworfene Qualitätsmodell bestehen natürlicherweise auch Qualitätsansprüche, welche in folgendem Kapitel beschrieben werden.

2.4.3 Erwünschte Modelleigenschaften

„High Level“ Design

Das Referenzmodell sollte sich idealerweise auf die Bewertung der Entwurfsmerkmale auf höheren Abstraktionsniveaus beschränken. Die Betrachtung dieser Merkmale bringt den Vorteil, dass sie schon in frühen Entwicklungsphasen messbar sind. Das können zum Beispiel Klassendiagramme sein.

Anwendungsempfehlungen

Es sollte eine klare Definition für die Bewertungsziele und Qualitätseigenschaften geben, welche durch das Modell festgelegt sind. Eine Einführung von Metriken ohne die Aussage über deren Einsatz zur Qualitätsbeurteilung verringert den Gebrauchswert des Modells.

Metrikdefinition

Die im Modell eingesetzten Metriken sollten genau definiert sein. Ist dies nicht der Fall, wird es, aufgrund der Ungenauigkeit, zu Fragen bei der Anwendung kommen.

Zum Beispiel könnten die Anzahl der Programmzeilen (*Lines of code*) auf verschiedenste Art und Weise interpretiert werden. Es können alle Zeilen, physikalische Zeilen, Anzahl der Anweisungen, usw. gemeint sein. Es ist zu erkennen, je exakter die Definition, desto weniger Unklarheiten entstehen bei der Anwendung.

Abhängigkeitsdefinition

Eine möglichst formale Definition der Abhängigkeiten von Messergebnissen zu korrelierenden Qualitätsmerkmalen erhöht den Modellgebrauchswert erheblich. Das können Schwellwerte sein, welche Messergebnisse mit definierten Qualitätscharakteristika verbinden. Auch könnten das Detektionsstrategien sein, welche eine Mustererkennung über eine reglementierte Auswertung von Messzusammenhängen beschreiben.

Interpretationsvorgaben

Um eine greifbare Aussage aus den gewonnenen Daten ziehen zu können, sind Vorgaben zur Interpretation eine erwünschte Eigenschaft eines Qualitätsmodells. Diese könnten von textueller bis hin zu visueller Aufbereitung reichen und den Zugang zu den Ergebnissen entscheidend verbessern.

Validierung

Die Grundlage eines Qualitätsmodells sind immer die auf Erfahrung beruhenden Meinungen der Autoren. Stellen diese die alleinige Basis des Modells, so besteht allerdings Spielraum für Zweifel an deren Richtigkeit. Es ist also erstrebenswert, ein Modell durch eine empirische Validierung zu festigen. Existieren erst einmal Ergebnisse aus der Anwendung des Modells, können diese mit dessen Aussagen verglichen werden, und das Modell so validiert werden.

2.4.4 Gegenüberstellung

Die in Tabelle 2.1 aufgezeigte kurze Geschichte objektorientierter Metriken nennt auch die wichtigsten Qualitätsmodelle in diesem Kontext. Im Folgenden werden diese Modelle nach [EWEBBF04] und [Kee06] genauer beschrieben und um das OOMIP-Modell erweitert. Abschließend erfolgt eine tabellarische Gegenüberstellung anhand der in Kapitel 2.4.3 genannten Attribute, um diese für die Auswahl während der Umsetzung vorzubereiten.

MOOSE

Die „Metric Suite for Object Oriented Design“ von Chidamber und Kemerer [SRK94] führte 1994 sechs Metriken im objektorientierten Kontext ein. Diese Arbeit gilt als ein wichtiger Wegbereiter für die darauf folgenden Qualitätsmodelle. Fehlt der Originalveröffentlichung noch eine Validierung, wurden diese in den Folgejahren unabhängig, z.B. als Indikator in Bezug auf die Fehlerdichte, durchgeführt.

Die vorgestellten Metriken sind:

- Weighted Method Per Class (WMC)
- The Depth of Inheritance Tree (DIT)
- The Number Of Children (NOC)
- The Coupling Between Object Classes (CBO)
- The Response For a Class (RFC)
- The Lack of Cohesion in Methods (LCOM)

Diese decken Vererbungs-, Komplexitäts-, Bindungs- und Kopplungseigenschaften ab.

LK OO Metriken

Im selben Jahr veröffentlichten Lorenz und Kidd ihre Metrik Sammlung [LK94]. Es handelt sich um wohl definierte und leicht anwendbare Metriken. Die Sammlung stand aber oft in der Kritik, nicht in ein hinreichend definiertes Qualitätsmodell eingebettet worden zu sein.

MOOD

1996 validierten Brito und Carpuca ihr im Vorjahr veröffentlichtes Qualitätsmodell „Metrics for Object Oriented Design“, genannt MOOD. 1998 folgte als Erweiterung die MOOD2 Suite. Die Betrachtung der objektorientierten Paradigmen steht hier im Vordergrund. Das sind die Kapselung, die Vererbung und der Polymorphismus.

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Polymorphism Factor (PF)
- Coupling Factor (CF)

Das Qualitätsmodell stellt die Metriken in funktionale Abhängigkeiten zu den Qualitätsmerkmalen, wie zum Beispiel der Fehlerdichte. Es folgte das MOODKIT als einfaches Werkzeug zur Extraktion der Metriken aus dem Quellcode mit der Unterstützung für C++ Code.

QMOOD

Das 2002 von Bansiya und Davis vorgestellte „Quality Model for Object-Oriented Design“ (QMOOD) bietet keine formale Definition von Metriken. Es ist ein hierarchisches Modell, bestehend aus folgenden Ebenen:

1. Eigenschaften der Designqualität (z.B. Wartbarkeit)
2. Objektorientiertes Design (z.B. Vererbung)
3. Objektorientierte Design-Metriken
4. Objektorientierte Design-Komponenten (z.B. Klassen)

Das QMOOD-Modell definiert zudem Korrelationen zwischen Metriken und Qualitätsattributen. Diese Indizes werden als *Quality Attribute Indices (QAI)* bezeichnet. Hierbei werden Eigenschaften wie Wiederverwendbarkeit oder Verständlichkeit betrachtet. Die Metriken sind allerdings nur textuell formuliert und lassen Mehrdeutigkeiten zu.

OOMIP

Lanza und Marinescu liefern 2006 die aktuellste Veröffentlichung im Kontext der objektorientierten Metriken und Qualitätsmodelle. Deren Name, „Object-Oriented Metrics in Practice“, soll hier mit OOMIP abgekürzt werden. Schwerpunkt der Arbeit bildet die Betrachtung von Metrikvisualisierungen und Detektionsstrategien zur Erkennung von Design-Disharmonien [Fow99]. Diese sollen im Kontext der Refaktorisierung eines SW-Systems angewandt werden. Weiter beschreiben Lanza und Marinescu auf Basis definierter Metriken, Korrelationen und Schwellwerten ein Qualitätsmodell zur Charakterisierung einer SW-Anwendung. Dabei werden die Messergebnisse strukturiert in der sogenannten Übersichtspyramide dargestellt. Der Betrachtungsfokus liegt hierbei auf den Eigenschaften Komplexität, Vererbung und Kopplung des Systems [LM06].

Vergleich

Ein Vergleich der oben vorgestellten Metriksammlungen und qualitativen Referenzmodelle findet in Tabelle 2.2 statt.

Eigenschaft	MOOSE	LK OO	MOOD	QMOOD	OOMIP
Nur "High level" Charakteristika	nein	ja	ja	ja	nein
Modellziele festgelegt	nein	nein	ja	ja	ja
Präzise Definition der Metriken	ja	ja	ja	ja	ja
Formale Beziehungsdefinition	nein	nein	ja	ja	ja
Ergebnis-interpretation	nein	nein	ja	ja	ja
Empirische Validierung	nein	nein	ja	ja	nein

Tabelle 2.2: Vergleich verschiedener OO-Qualitätsmodelle.

2.5 Messtheorie

2.5.1 Einleitung

Vor der Definition der gewünschten Messziele und der Auswahl geeigneter Metriken sollen folgend die im Kontext der Arbeit wichtigen Grundlagen der Messtheorie erläutert werden. Diese sind notwendige Bedingungen, um qualitativ hochwertige Ergebnisse zu erhalten. In Abbildung 2.9 werden die in diesem Zusammenhang wichtigen Beziehungen dargestellt [AKP02].

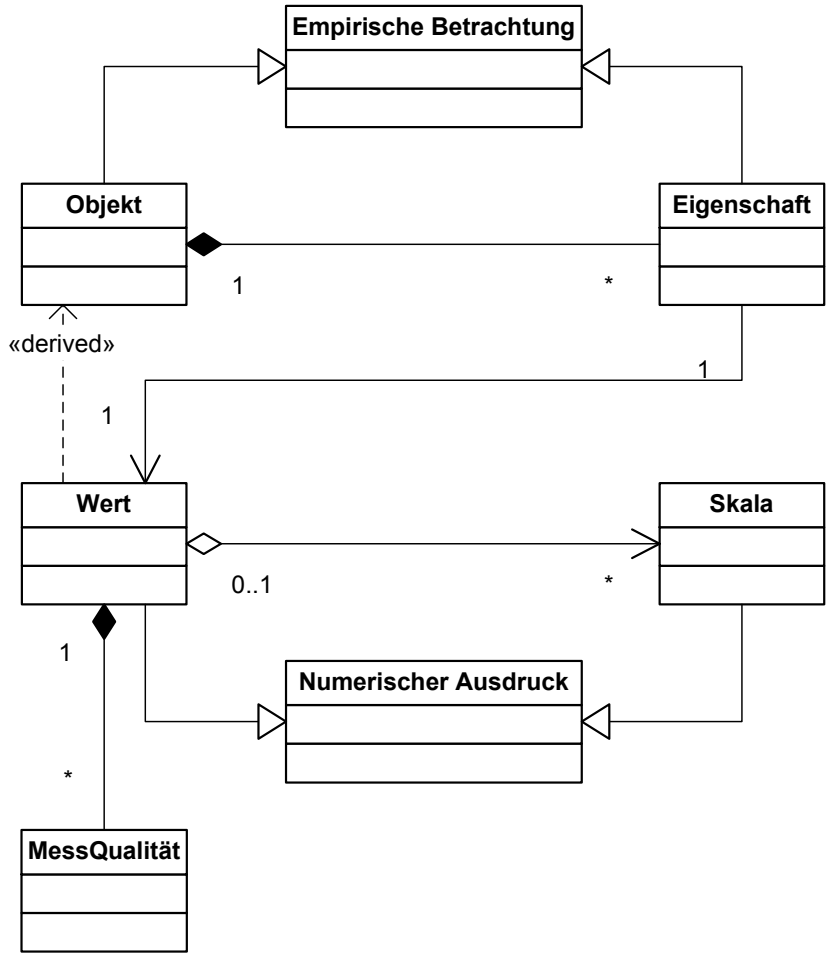


Abbildung 2.9: Beziehung im Kontext der Messung.

Soll ein Objekt der realen Welt in einem Zahlenbereich abgebildet werden, muss dafür eine geeignete Skalierung gewählt werden. Dabei kann aus vier Graden der Einteilung [Rie96] gewählt werden. Es handelt sich hierbei um die Nominal-, die Ordinal-, die Intervall- und die Verhältnisskalierung.

Weiter stellt sich vor einer Messung die Frage, wie die Qualität der Messergebnisse ausfallen wird. Neben der Robustheit und den praktischen Aspekten [HS96] einer Messung sind die beiden wichtigsten zu betrachtenden Eigenschaften die Verlässlichkeit und die Gültigkeit [Rie96], [FP96], [HS96].

2.5.2 Skalierung

Nominalskalierung

Die Klassifizierung ist der einfachste Grad einer Messeinteilung. Es folgt dazu als Beispiel eine Betrachtung von Berufsgruppen. Diese können in Bäcker/in, Politiker/in, Tischler/in, Arzt/Ärztin etc. klassifiziert werden. Als Anforderung an diese Einteilung ist zu beachten, dass die Klassen vollständig die möglichen Kategorien der betrachteten Eigenschaft abdecken. Weiter muss die Einteilung gewährleisten, dass ein zu klassifizierendes Objekt nur einer Kategorie zuordenbar ist. Im Beispiel sollte so eine Person nur einer Berufsgruppe zuordenbar sein. Damit werden die notwendigen Minimalbedingungen für die Anwendung zur statistischen Analyse eingehalten. Die Kategorien stehen dabei in keinem bewertbaren Zusammenhang. Es kann als Beispiel anhand der Skalierung keine Aussage darüber getätigt werden, ob ein Bäcker mehr arbeitet als ein Arzt.

Ordinalskalierung

Ergänzt man die Klassifizierung durch eine Ordnung, erhält man eine ordinale Messeinteilung. Nun können Aussagen über die Anordnung der Kategorien angestellt werden. Als Beispiel eine Betrachtung von Transportmitteln anhand ihres durchschnittlichen Co2-Ausstoßes auf 100km pro Tonne im Gütertransport. Beginnend mit den geringsten Emissionen kann wie folgt kategorisiert werden: Schiff, Zug, Auto, Flugzeug. Es wird dabei allerdings keine Aussage über das Ausmaß getroffen. Es kann nur gesagt werden, dass ein Schiff weniger Co2 freisetzt als ein Flugzeug. Wie groß die Differenz ist, kann dabei nicht erkannt werden. Im mathematischen Kontext kann über ordinal eingeteilte Werte nur ausgesagt werden, dass diese größer, kleiner oder gleich sind. Operation wie z.B. die Addition sind nicht anwendbar.

Intervallskalierung

Addition und Subtraktion können angewandt werden, sobald eine eindeutige Bestimmung der Messpunkte eingeführt wird. Im Bezug auf das oben eingeführte Beispiel der Transportmittel soll jetzt von folgendem ausgegangen werden: das Schiff emittiert 1 kg und das Flugzeug 50 kg Co2 auf 100 km pro Tonne. Es kann jetzt die Aussage getroffen werden, dass das Flugzeug 49 kg Co2 mehr freigibt als das Schiff. Dabei muss zur Bestimmung eine klar definierte und allgemein gebräuchliche Messeinheit als Basis angenommen werden. Im Beispiel sind das die Kilogramm.

Verhältnisskalierung

Eine Intervallskalierung wird durch das Auffinden eines absoluten Nullpunkts zur Verhältnisskalierung. Die Operationen Multiplikation und Division können angewandt werden.

Eigenschaften

Vergleicht man die Grade der Skalierung, ist eine Hierarchie der Spezialisierung erkennbar.

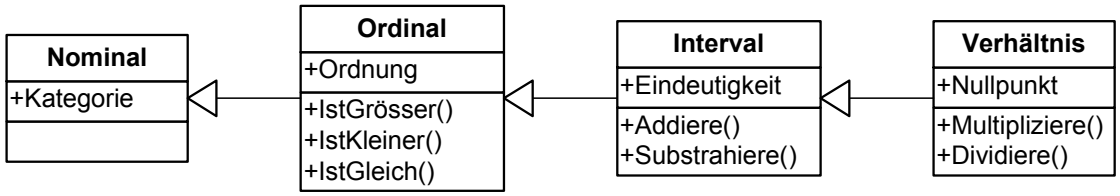


Abbildung 2.10: Hierarchie der Skalierungsklassen.

Je stärker der angewandte Grad der Spezialisierung, desto besser die Anwendbarkeit statistischer Analyse darauf. Es ist daher immer sinnvoll, eine Skalierung mit möglichst hoher Spezialisierung zu wählen, sofern das der Kontext zulässt.

2.5.3 Messqualität

Zuverlässigkeit

Die Zuverlässigkeit ist eine notwendige Bedingung an die Konsistenz der Messung. Grundsätzlich kann davon ausgegangen werden, dass es bei jeder Messung zu einem systematischen Messfehler kommt. Dass dieser möglichst klein ausfällt, wird dadurch erreicht, dass die Messwerte stabil und gleichwertig gehalten werden [HS96]. Ein hohes Maß an Stabilität ist erreicht, sobald unter identischen Messbedingungen möglichst identische Werte gemessen werden. Äquivalente Ergebnisse werden erzielt, sobald die Messwerte unabhängig vom Messgegenstand sind. Eine hohe Verlässlichkeit wird durch eine möglichst präzise Definition der Messbedingungen erreicht. Als Beispiel soll die durchschnittliche Jahrestemperatur einer Region ermitteln werden. Das Verbessern der Zuverlässigkeit kann durch die Vorgabe folgender Bedingungen erreicht werden: Messzeitpunkt, der Messort, die Anforderungen an denjenigen, der die Messung durchführt, die Skalierung, etc. [Rie96].

Gültigkeit

Die Gültigkeit einer Messung drückt sich in der Übereinstimmung der Ergebnisse mit der Bedeutung der für die Messung entworfenen Konzepte aus. Bei der Gültigkeit stellt sich die Frage, ob das Messergebnis dem entspricht, was gemessen werden soll [Rie96]. Es soll das oben genannte Beispiel zur Jahrestemperatur weitergeführt werden. Für die Temperaturbestimmung bedient sich ein europäischer Meteorologe bei einem amerikanischen Kollegen mit Messergebnissen. Dabei legt er fälschlicherweise die Einheit Grad Celsius an die in Grad Fahrenheit gemessene Jahrestemperatur an. Die amerikanischen Kollegen waren sehr genau und das Ergebnis ist zuverlässig. Der Umrechnungsfehler macht es aber ungültig. Das Beispiel zeigt, dass man ungültige Messergebnisse auch noch nach der Messung mittels Umrechnung zu gültigen Werten umformulieren kann.

Zusammenhang

Eine grafische Darstellung des Zusammenhangs von Gültigkeit und Zuverlässigkeit wird in der folgenden Darstellung gezeigt [Rie96].

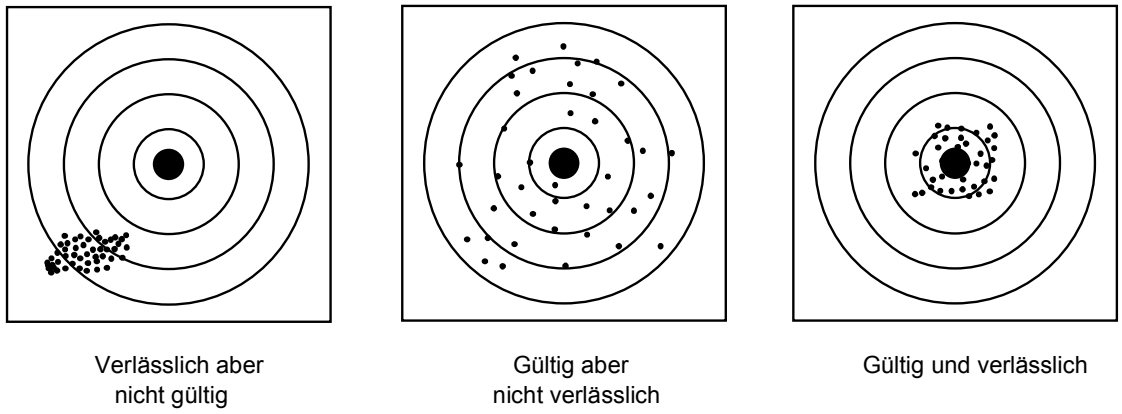


Abbildung 2.11: Zusammenhang von Verlässlichkeit und Gültigkeit.

Kapitel 3

Umsetzung

3.1 Einleitung

Auf Basis des vorangegangenen Kapitels 2 wird jetzt die Umsetzung der Messanwendung beschrieben. Eine Betrachtung der Anforderungen und Anwendungsfälle führt zur Auswahl der geeigneten Messmethoden. Es folgt einer Erläuterung der für den MDSD-Kontext notwendigen Anpassungen. Das Vorgehen zur Interpretation der gewählten Methodik schließt das Kapitel ab.

3.2 Anforderungen

Nr.	Anforderung
1	Eine Auflistung der Zählungen wichtigster Modellelemente muss dargestellt werden.
2	Ein Querschnittsüberblick zur Modellgüte muss dargestellt werden.
3	Die psychologische Komplexität von Diagrammen soll dargestellt werden.
4	Die Messanwendung muss als Rhapsody-Plugin in Java mittels der Rhapsody-Java-API realisiert werden.
5	Die Messanwendung soll zur kontinuierlichen Überwachung automatisiert aufgerufen werden können.

Tabelle 3.1: Anforderungen an die Messanwendung.

Für die Auswahl der Messmethodik dienen die in Tabelle 3.1 gelisteten Anforderungen der Messanwendung. Daraus lassen sich die in Abbildung 3.1 gezeigten Anwendungsfälle modellieren.

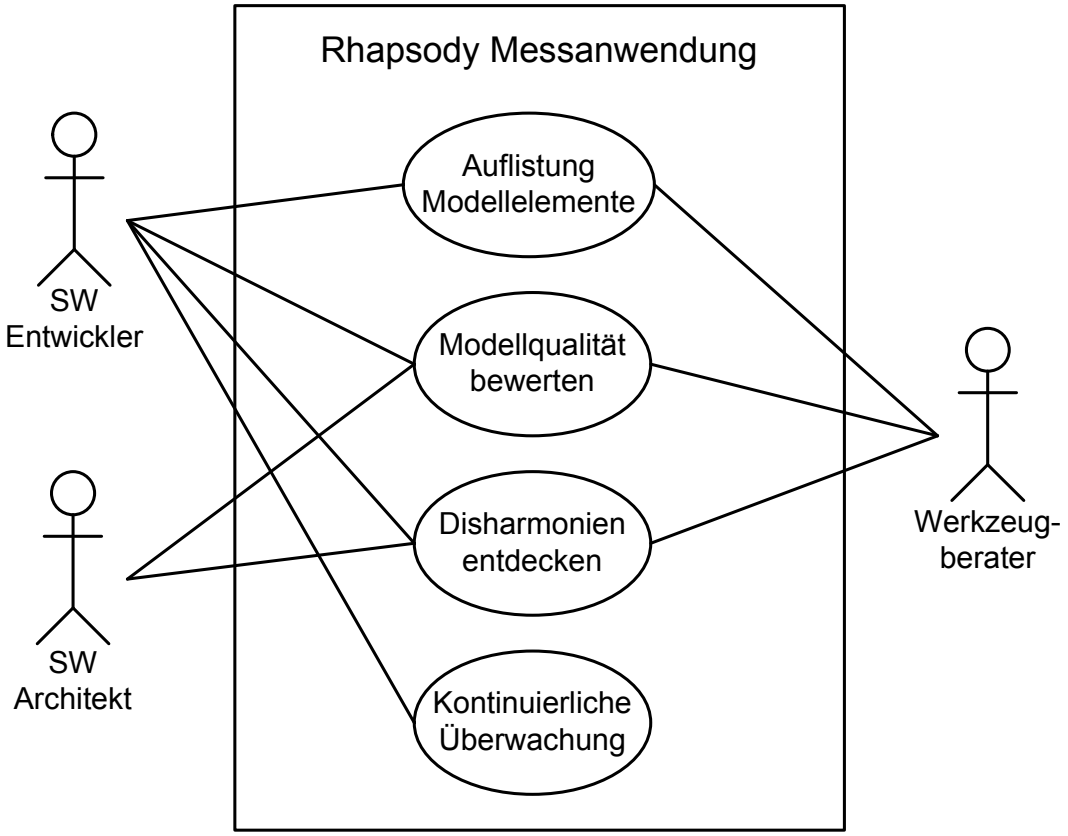


Abbildung 3.1: Anwendungsfälle der Messanwendung.

Die Auflistung der Modellelemente und die kontinuierliche Überwachung ist unabhängig von Messmethode und Qualitätsmodellen zu betrachten und muss „nur“ umgesetzt werden. Dagegen fordert die Darstellung der Modellqualität und der Komplexität von Diagrammen weitere Überlegungen zur Umsetzung. Für diese beiden Punkte sollen im nächsten Kapitel geeignete Methoden gefunden werden.

3.3 Auswahl der Messmethoden

Wie in Kapitel 2.2 beschrieben, haben wir es mit einem formalen Modell zu tun, das mit Code-Artefakten angereichert ist. Die Messung beschränkt sich also nicht auf ein Modell allein, sondern bezieht sich zudem auf eine Implementierungssprache. Zudem impliziert die kontinuierliche Überwachung die Notwendigkeit für eine effiziente Messmethode, um den Entwicklungsfluss nicht zu beeinträchtigen. Die unterschiedlichen Akteure machen es weiter erforderlich, bei der Auswahl auf einfache und verständliche Verfahren zu setzen, um die Kommunikation und den Ad-hoc-Gebrauchswert zu fördern. Aufgrund dieser Betrachtung sowie dem Vergleich der Qualitätsmodelle und deren Methoden aus Kapitel 2.4, wird hier folgende Auswahl getroffen. Zur Darstellung der Modellqualität soll die von [LM06] in OOMIP vorgeschlagene Vorgehensweise der Übersichtspyramide umgesetzt werden. Für die Abbildung komplexer Diagramme werden zudem die in Tabelle 3.2 gelisteten Methoden gewählt.

Anwendungsfall	**Gewählte Umsetzung**
Darstellung	An MDSD angepasste Übersichtspyramide nach [LM06].
Modellqualität	
Komplexität von Objektmodelldigrammen darstellen	Die „Siebener Regel“ nach [Mil56].
Komplexität von Aktivitätsdiagrammen und Zustandsautomaten darstellen	McCabe Komplexität nach [McC76].

Tabelle 3.2: Für die Anwendung gewählte Messmethoden.

3.4 Metrikdefinitionen

3.4.1 Übersichtspyramide für MDSD

Die für die Darstellung der Modellqualität gewählte Übersichtspyramide nach [LM06] teilt sich in die drei in Abbildung 3.2 gezeigten Bereiche auf. Diese stellen die ihren Fokus betreffenden Metriken dar. Die Bereiche für Größe und Kopplung besitzen zudem Werte, die sich aus berechneten Proportionen der Metriken bilden. Der Vorteil der Proportionen gegenüber den direkten Messwerten ist deren Unabhängigkeit von der Größe der gemessenen Anwendung. Dadurch werden diese vergleichbar und es lassen sich Schwellwerte finden. Die Schwellwerte ermöglichen die Interpretation der Proportionen in drei geordnete Bereiche. Diese ordinale Skalierung wird bei den Proportionen und Vererbungsmetriken durch eine farbige Markierung dargestellt. Die Schwellwerte werden auf der Grundlage einer statistischen Erhebung gebildet (Kapitel 5) und teilen die Proportionen in niedrige, durchschnittliche und hohe Werte ein.

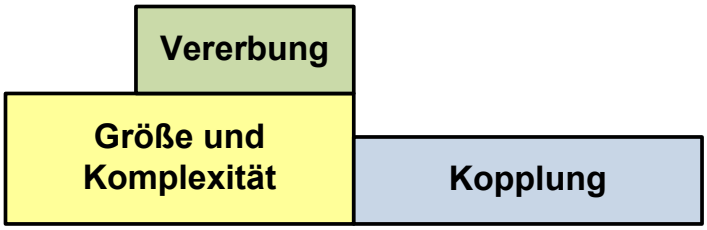


Abbildung 3.2: Aufteilung der Übersichtspyramide.

Die ursprünglich vorgeschlagene Pyramide kann nicht sinnvoll auf die MDSD angewandt werden und benötigt eine Anpassung. Besonders die Interpretationen der Proportionen müssen teilweise neu formuliert werden. Der Grund ist die zu einer vollständig handgeschriebenen Anwendung unterschiedliche Verteilung der Quelltext-Artefakte. Das bloße Zählen der Methoden im Originalansatz wird um das Zählen der Artefakte erweitert. Der Bereich Kopplung wird in diesem Ansatz besonders an die im Kapitel 2.2.3 erwähnte ereignisgetriebene Architektur angepasst. Die angepasste Pyramide ist vollständig in Abbildung 3.3 dargestellt.

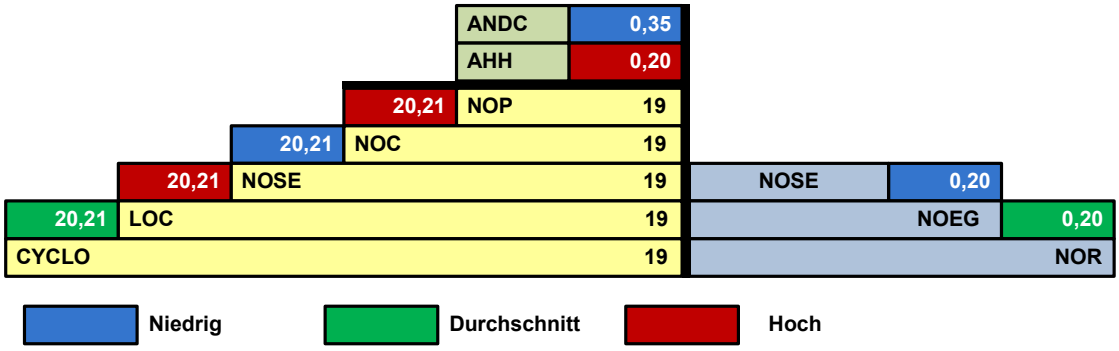


Abbildung 3.3: Übersichtspyramide des Modells.

Die verwendeten Metriken und Proportionen sind in den Tabellen 3.3 und 3.4 gelistet. Deren Definition wird in Kapitel 3.4.2 ff. beschrieben.

Abkürzung	**Lang**	**Bedeutung**
NOP	Number of packages	Anzahl der Pakete.
NOC	Number of classes	Anzahl der Klassen.
NOSE	Number of source elements	Anzahl der Code-Artefakte.
LOC	Lines of code.	Anzahl der Programmzeilen.
CYCLO	Cyclomatic complexity	Zyklomatische Komplexität eines Zustandsautomaten oder Aktivitätsdigramms.
NOEG	Number of event generations	Anzahl der generierten Ereignisse.
NOR	Number of relations	Anzahl der Assoziationen einer Klasse.
ANDC	Average number of derived classes	Durchschnittsanzahl der vererbten Klassen.
AHH	Average hierarchy height	Durchschnittliche Höhe der Vererbungshierarchie.

Tabelle 3.3: Metriken der angepassten Übersichtspyramide.

Proportion	Durchschnittliche Anzahl
NOC/NOP	Klassen pro Paket.
NOSE/NOC	Methoden pro Klasse.
LOC/NOSE	Zeilen pro Quelltextartefakt.
CYCLO/LOC	Modellkomplexität pro Quelltextzeilen.
NOEG/NOSE	Generierte Ereignisse pro Quelltextartefakt.
NOR/NOEG	Relationen pro generiertem Ereignis.

Tabelle 3.4: Proportionen der angepassten Übersichtspyramide.

3.4.2 Größen- und Komplexitätsmetriken

Anzahl der Pakete

Ein Paket ist das zentrale Strukturierungselement eines Rhapsody-Modells. Es entspricht dem Paket der UML und dient der Einteilung des Modells in funktionale Bereiche oder Teilsysteme, unter welchen die logischen Artefakte des Modells Platz finden. Sie selbst haben keinen Einfluss auf das Verhalten des Systems, unterteilen es aber auf höchster Ebene.

$$NOP = \sum_{m=0}^{Modellelemente} IstPaket(m) \tag{3.1}$$

Es werden alle Pakete des Projekts oder des betrachteten Modellelements gezählt.

Anzahl der Klassen

Die Klassen sind die Kerne des Modells und repräsentieren dessen statische Struktur. Sie können in Klassendiagrammen (Objektmodelldiagrammen) dargestellt werden.

$$NOC = \sum_{m=0}^{Modellelemente} IstKlasse(m) \tag{3.2}$$

Bei der Zählung der Klassen werden Klassen und Interface-Klassen gleichbehandelt.

Anzahl der Quelltext-Artefakte

Die ursprüngliche Messung der Methoden nach [LM06] muss im Kontext der MDSD erweitert werden. Wie in Kapitel 2.2.2 beschrieben, wird das Modell erst durch Code-Artefakte vollständig formal beschrieben, und kann somit für das erwünschte Zielsystem generiert werden. Diese Code-Artefakte beschränken sich daher nicht nur auf Methoden, sondern sind in weiteren Modellelementen formuliert. Diese sind in der folgenden Tabelle beschrieben.

Element	Quelltextartefakt
Action	Aktion, welche beim Betreten, bei Nachrichtenverarbeitung innerhalb eines Zustands oder bei Verlassen ausgeführt werden soll. Weiter kann die Aktion auch den Quelltext enthalten, welcher beim Auslösen eines Übergangs durchgeführt werden soll.
Guard	Bedingung für einen Transitionsübergang.
Operation	Quelltext der Methode einer Klasse.
Trigger	Quelltext des Auslösers einer Transition.

Tabelle 3.5: Quelltextartefakte eines Rhapsody-Modells.

Diese Elemente werden gleichbehandelt und als jeweils ein Code-Element gezählt.

$$NOSE = \sum_{m=0}^{Modellelemente} IstCodeArtefakt(m) \tag{3.3}$$

Anzahl der Programmzeilen

Die Anzahl der Programmzeilen soll für die Implementierungssprachen C, C++ und Java gemessen werden. Es werden die physikalischen Zeilen gezählt. Eine physikalische Code-Zeile ist definiert als Zeile, welche mit einer neuen Zeile oder mit dem Ende des Artefakts endet. Zudem enthält sie mindestens ein Zeichen, welches kein Leerzeichen und kein Kommentarzeichen ist. Zeilen, die ausschließliche Leerzeichen oder Tabulatoren enthalten, werden nicht gezählt. Kommentarbegrenzungen werden als Kommentarzeichen betrachtet [Whe09].

$$LOC = \sum_{m=0}^{Code-Artefakte} PhysikalischeZeilenanzahl(m) \tag{3.4}$$

Im folgenden kurzen Quelltextbeispiel können so sechs Programmzeilen gezählt werden. Die reine Kommentarzeile und die Leerzeile werden nicht gezählt.

```
#include <stdio.h>

int main(void)
{
    /* Dies ist ein Kommentar. */
    printf("Hallo Welt!\n");
    return 0; // Rückgabe
}
```

Die Definition der Messung bringt naturgemäß eine verringerte Zuverlässigkeit (s. Kapitel 2.5.3) gegenüber dem Zählen von logischen Anweisungen. Diese ist aber vernachlässigbar, da sie in der Praxis wenig Einfluss auf den Gebrauchswert dieser Metrik besitzt (vgl. hierzu auch beispielsweise [Whe01]). Zudem ist das Zählen von physikalischen Programmzeilen weitaus einfacher und effizienter zu implementieren als das Zählen von Anweisungen.

Zyklomatische Komplexität

Gegenüber [LM06] sollen nicht die Komplexität der Kontrollflüsse des Quelltextes gemessen werden, sondern die der modellierten Graphen. Dazu folgende Definition.

Die zyklomatische Zahl nach McCabe [McC76] gibt die Anzahl der Basispfade eines Graphen an [Hop08]:

$$v(G) = e - n + 2p \tag{3.5}$$

G: Kontrollflussgraph
e: Anzahl der Kanten bzw. Zweige
n: Anzahl der Knoten bzw. Anweisungen
p: Anzahl der zusammenhängenden Komponenten

Bei dieser Metrik sollen nicht zusammenhängende Kontrollflussgraphen gemessen werden. Für p können wir daher von einem Wert von eins ausgehen und auf folgende Definition der zyklomatischen Zahl vereinfachen [Hop08]:

$$v(G) = e - n + 2 \tag{3.6}$$

Für die Messung der McCabe-Komplexität [McC76] im Modell werden zwei relevante Formen der Kontrollflussgraphen betrachtet. Dabei handelt es sich um das Aktivitätsdiagramm und den Zustandsautomaten nach der UML 2. Beide Diagramme beschreiben das Verhalten einer Klasse und beeinflussen direkt den generierten Code.

$$CYCLO = \sum_{m=0}^{Kontrollflussgraphen} v(m) \tag{3.7}$$

Während die Aktivitätsdiagramme „flach“ zu modellieren sind, handelt sich bei den Zustandsautomaten um Automaten nach Harel [Har87]. Für beide sollen alle Elemente des Graphen, welche nicht Transitionen oder Kontrollflüsse sind, als Knoten gezählt werden. Diese sind folgend aufgelistet:

Zustandsautomat	**Aktivitätsdiagramm**
Zustände	Aktivitäten
Parallelzustände	Verzweigungs- und Verbindungskonten
Unterzustände	Startknoten
Bedingungen	Endknoten
Historien-Verbinder	Sendeaktionen
Synchronisations- und Parallelisierungsknoten	
Sendeaktionen	
Reaktionen innerhalb eines Zustands	
Verbindungskonten	
Terminatoren	

Tabelle 3.6: Knotenelemente der Modell-Kontrollflussgraphen.

3.4.3 Kopplungsmetriken

Anzahl der Ereignisgenerierungen

Um Kopplungseigenschaften darstellen zu können, soll hier auf die in Kapitel 2.2.3 eingeführte Besonderheit der ereignisgetriebenen Architektur der betrachteten Modelle eingegangen werden. Diese unterscheidet sich von der im Ursprung vorgeschlagenen Betrachtung objektorientierter Kopplung durch die Messung der Anzahl der eindeutigen Methodenaufrufe. Im Modell können hingegen Ereignisse spezifiziert werden, welche synchron oder asynchron versendet werden. Die Sendeaufrufe werden in den handgeschriebenen Code-Artefakten erzeugt.

$$NOEG = \sum_{m=0}^{Code-Artefakt} \sum_{n=0}^{Ereignisname} IstEnthalten(m,n) \tag{3.8}$$

Ist ein Ereignis in einem Code-Artefakt mindestens einmal enthalten, wird der Messwert inkrementiert. Da die hier untersuchten Sendeaufrufe handgeschrieben sind, können bei der Entwicklung beliebige Konventionen angewandt werden. Diese Tatsache schränkt die Zuverlässigkeit der Metrik ein. Die Aufrufe könnten z.B. in Methoden gekapselt sein und deren Ereignisnamen verbergen. Es ist aber davon auszugehen, dass die Abweichungen verhältnismäßig selten auftreten, sodass die Zuverlässigkeit dieser Metrik nicht sehr stark beeinträchtigt wird.

Anzahl der Assoziationen

Die von [LM06] vorgeschlagen FANOUT-Metrik nach [LK94] zählt die Klassen, welche durch die ursprüngliche Kopplungsmetrik bewertete Methoden aufrufen. Diese soll hier durch die Anzahl der bekannten Kommunikationswege einer Klasse ersetzt werden. Die oben beschriebenen Sendeaufrufe benötigen die Angabe des Empfängers, der damit dem Sender bekannt sein muss. Damit soll hier davon ausgegangen werden, dass diese Assoziationen auch zur ereignisgetriebenen Kommunikation genutzt werden. Die Zählung ist unabhängig davon, ob es sich um binäre Assoziationen, Aggregationen oder Kompositionen handelt.

$$NOR = \sum_{m=0}^{Klassen} AnzahlAssoziationen(m) \tag{3.9}$$

3.4.4 Vererbungsmetriken

Durchschnittsanzahl der vererbten Klassen

Diese Metrik kann auch als die Anzahl der direkten Nachkommen oder Unterklassen beschrieben werden. Interface-Klassen werden dabei nicht mitgezählt. Besitzt eine Klasse keine Nachkommen, so wird diese mit 0 gezählt [LM06].

$$ANDC = \frac{\sum_{m=0}^{Klassen} AnzahlDirekteNachkommen(m)}{Klassen} \tag{3.10}$$

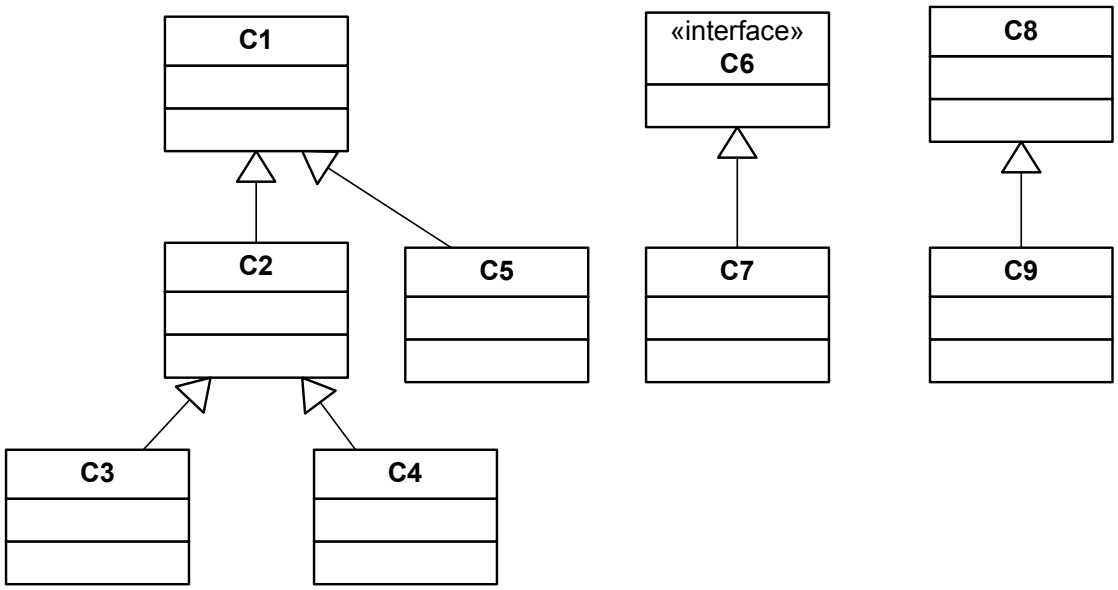


Abbildung 3.4: Beispiel einer Vererbungshierarchie.

Das Metrikergebnis für die in Abbildung 3.4 gezeigte Hierarchie ist als Beispiel:

$$ANDC = \frac{3 \cdot 0 + 1 \cdot 1 + 2 \cdot 2}{8} = 0.625$$

Durchschnittliche Höhe der Vererbungshierarchie

Eine Erweiterung der Metrik zur Höhe des Vererbungsbaums *HIT* aus [FP96] soll hier angewandt werden [LM06]. Sie kann beschrieben werden als durchschnittliche Maximallänge des Pfades der Vererbungshöhe bis zu ihrem tiefsten Nachkömmling. Formal kann diese so ausgedrückt werden [Hau09]:

$$AHH = \frac{\sum_{m=0}^{AnzahlBasisklassen} HIT(m)}{AnzahlBasisklassen} \tag{3.11}$$

mit

$$\text{HIT(Basisklasse)} = 0 \tag{3.12}$$

Für das Beispiel in Abbildung 3.4 ist das Metrikergebnis:

$$AHH = \frac{2+0+1}{3} = 1$$

3.5 Interpretation der Pyramide

3.5.1 Größen- und Komplexitätsproportionen

Klassen pro Paket

Diese Proportion hilft bei der Bewertung der Strukturierung des gemessenen Systems auf höchster Ebene. Es wird eine Aussage darüber getroffen, ob die Struktur eher fein oder grob ist.

Quelltext-Artefakt pro Klasse

Der Strukturierungsgrad der Klassen im System kann durch die durchschnittliche Anzahl der Quelltext-Artefakte pro Klasse erkannt werden. Hohe Werte können hier auf fehlende Klassen oder nicht refaktorisierte Kontrollflussgraphen des Modells hindeuten.

Zeilen pro Quelltextartefakt

Die Verteilung des Quelltextes über die Artefakte wird durch diese Proportion dargestellt. Hohe Werte können auf eine Tendenz zu „großen“ Quelltext-Artefakten hinweisen.

Modellkomplexität pro Zeile Code

Die Modellkomplexität pro Zeile Code kann einen ersten Eindruck darüber geben, wie stark die modellierte Verhaltensmodellierung gegenüber handgeschriebenem Quelltext genutzt wird. Hohe Werte geben einen ersten Hinweis darauf, dass die Modellmöglichkeiten zu Verhaltensmodellierung eher konservativ genutzt werden.

3.5.2 Kopplungsproportionen

Generierte Ereignisse pro Quelltextartefakt

Dieses Verhältnis zeigt die Qualität der Kollaboration der Objekte im ereignisgetriebenen Kontext. Hohe Werte können ein Zeichen für eine übermäßige Kopplung auf dieser Ebene der Architektur sein.

Relationen pro generiertem Ereignis

Die Relationen pro generiertem Ereignis können ein Hinweis darauf sein, wie schwach oder stark die Bindung von Klassen im Bezug auf die ereignisgetriebenen Kopplungsausbreitung [Hau09] ist. Hohe Werte können eine große Interaktion vieler unterschiedlicher Klassen aufzeigen.

3.5.3 Vererbungsproportionen

Durchschnittsanzahl der vererbten Klassen

Die Durchschnittsanzahl der vererbten Klassen kann einen Eindruck über die „Breite“ der Vererbungshierarchie geben.

Durchschnittliche Höhe der Vererbungshierarchie

Mithilfe dieser Proportion lässt sich ein erster Eindruck für die „Höhe“ der Vererbungshierarchie bekommen.

3.6 Detektionsstrategien

3.6.1 McCabe

Die zyklomatische Komplexität eignet sich besonders zur Detektion zu komplexer Kontrollflussgraphen [AV04]. Die hier vorgestellte Methode kann z.B. während einer Refaktorisierung angewandt werden. Die Elemente mit der größten Komplexität werden zuerst betrachtet und gegebenenfalls refaktorisiert [Fow99]. Daher soll nicht mit einem feststehenden Schwellwert gearbeitet werden. Das Auffinden soll durch das beliebige Anpassen der größten Werte ermöglicht werden. Die Komplexität ermittelt sich wie in Kapitel 3.4.2 beschrieben für die Kontrollflussgraphen des Modells.

3.6.2 Millersche Zahl

Die Millersche Zahl oder auch „Siebener Regel“ besagt, dass das menschliche Kurzzeitgedächtnis nur 7±2 Informationseinheiten gleichzeitig verarbeiten kann [Mil56]. Dieses Phänomen soll die konzeptionelle Grundlage für das Detektieren von komplexen Objektmodelldiagrammen sein. Wie bei der oben genannten Strategie soll aber auch hier auf einen festen Schwellwert verzichtet werden. Der Ansatz sieht wiederholt eine flexible Anpassung der oberen Komplexitätsgrenze vor. Die Komplexität für die Diagramme wird aus der summierten Anzahl der darauf dargestellten Informationseinheiten berechnet. Als relevante Informationseinheiten sollen dabei die durch Klassen, Instanzen, Module und Dateien dargestellten Entitäten betrachtet werden.

Kapitel 4

Softwarearchitektur

4.1 Einleitung

Das im vorangegangenen Kapitel 3 gezeigte konzeptionelle Vorgehen wird in diesem Kapitel in eine konkrete Software-Architektur umgesetzt. Zu Beginn wird die Verteilung des Java Metrik Plugins für Rhapsody und deren Kommunikationsverbindung erläutert. Es folgt eine Darstellung der Architektur- und Design-Entwurfskonzepte. Die groben Konzepte werden dann in einer detaillierten Darstellung der Implementierungsstruktur, anhand von Klassendiagrammen, verfeinert. Dabei werden nur die für das Verständnis wichtigsten Methoden besprochen. Eine genaue Beschreibung der API und aller öffentlichen Methoden und Attribute der Messanwendung ist in Anhang E zu finden. Das zugehörige Anwendungsverhalten und das Zusammenspiel der Klassen wird mittels Sequenzen veranschaulicht. Hier wird auch der Aufruf über die API-Callback-Methoden seitens Rhapsody modelliert.

Es folgt eine Vorstellung der verwendeten Bibliotheken und fremder Quelltext-Fragmente, die ergänzend zum Standard Java Rahmen ergänzend benutzt werden. Abschließend wird auf die während der Implementierung angewandten Software-Qualitätsmaßnahmen und Werkzeuge eingegangen.

4.2 Struktur

Um einen ersten Überblick auf hoher Abstraktionsebene zu bekommen, wird die Verteilung der Artefakte und Komponenten der Anwendung in Abbildung 4.1 gezeigt.

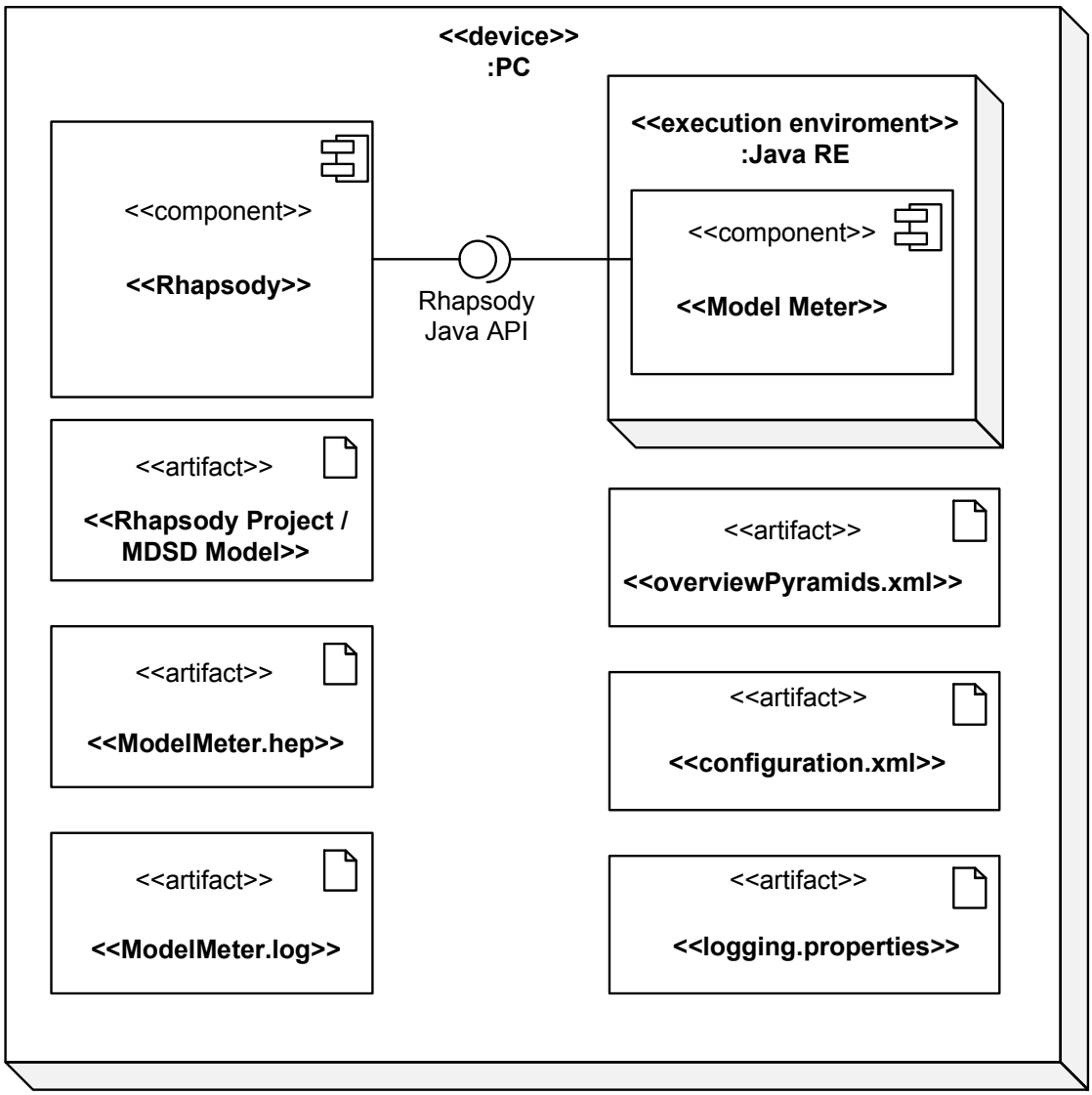


Abbildung 4.1: Verteilung der Messanwendung.

Auf einem Rechner findet die Kommunikation zwischen einer Instanz des Werkzeugs Rhapsody und des in einer virtuellen Java Maschine laufenden Plugin über die Rhapsody eigene API statt. Die dargestellten Artefakte bestehen aus dem zu messenden Rhapsody-Projekt, sowie den Konfigurationsdaten für die Messanwendung, und werden in Tabelle 4.1 beschrieben.

Artefakt	**Funktion**
configuration.xml	Allgemeine Konfiguration (Build-Nummer, aktuell gewähltes Referenzmodell).
logging.properties	Konfiguration des Protokollierungsverhaltens.
overviewPyramids.xml	Definition von Referenzmodellen für die Berechnung einer Übersichtpyramide.
ModelMeter.hep	Steuerdatei zur Einbindung des Plugins in Rhapsody (Menüs, Trigger).
ModelMeter.log	Protokolldatei des Plugins für Informationen und Warnungen.
Rhapsody Project	Zu analysierendes MDSD-Modell Rhapsody's.

Tabelle 4.1: Funktion der Anwendungsartefakte.

4.3 Entwurfsmuster

4.3.1 Model-View-Controller

Aufgrund der Entscheidungen zur Umsetzung aus Kapitel 3 steht fest, dass die Datenhaltung, die Datenverarbeitung und eine grafische Darstellung der Messergebnisse implementiert werden müssen.

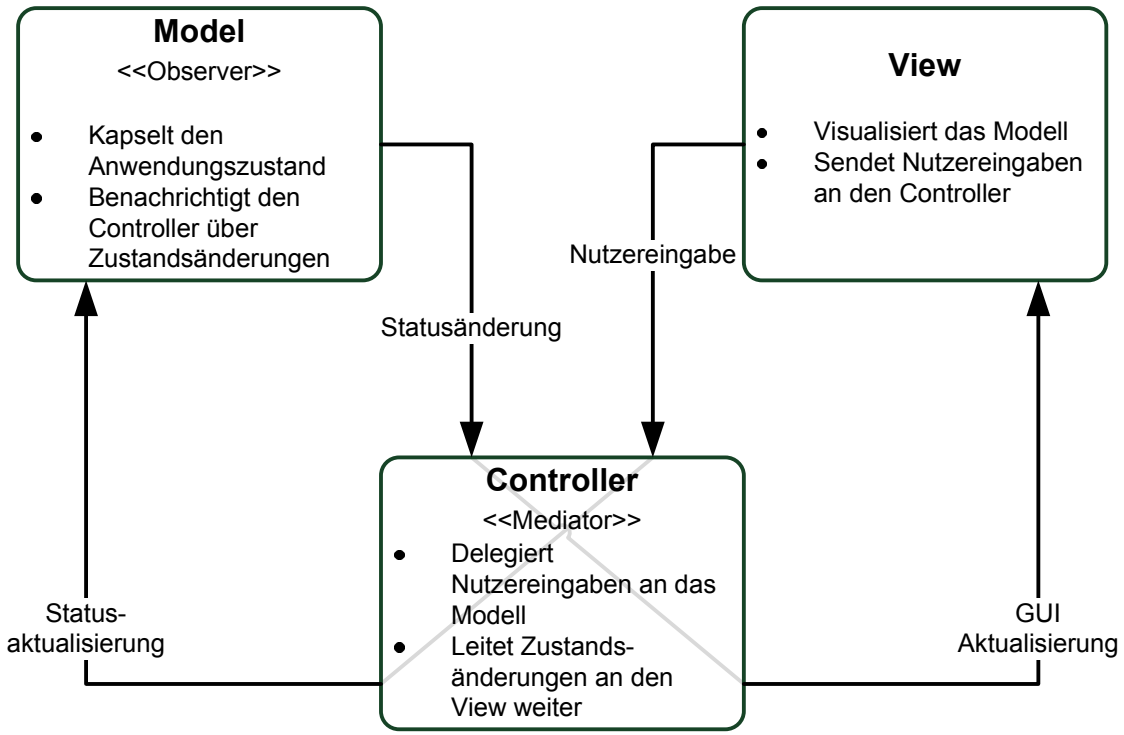


Abbildung 4.2: Die *Model-View-Controller* Architektur der Anwendung.

Eine dafür geeignete Architektur ist die in Abbildung 4.2 gezeigte Strukturierung mittels des *Model-View-Controller*-Musters (MVC). Diese spezielle Variante des Layerd-Musters [GMP08] setzt sich aus drei Schichten für Datenhaltung (*Model*), Programmlogik (*Controller*) und Präsentation (*View*) zusammen. Die strikte Trennung der Schichten bringt beim MVC-Muster den Vorteil, dass das *Model* die Datenhaltung und den größten Teil der Verarbeitung kapselt.

4.3.2 Observer

Die Schichten der MCV-Architektur werden mit einem ereignisgetriebenen Beobachter-Entwurfsmuster [GHJV94] lose gekoppelt. Durch das Beobachter-Muster ist es nicht notwendig, dass ein *Model* seine Präsentation kennt. Dadurch können beliebig viele *Views* *Model*-Daten abbilden. Weiter ist der einfache Austausch eines *Views* möglich, ohne das korrespondierende *Model* dafür ändern zu müssen.

4.3.3 Mediator

Wie in Abbildung 4.2 zu sehen ist, findet die Kommunikation in dieser Variante des MVC-Architekturmusters vollständig über den *Controller* statt. Dieses Verhalten entspricht dem Entwurfsmuster des Mediators (Vermittlers) [GHJV94]. Nutzereingaben der *Views* werden vom *Controller* an das entsprechende *Model* delegiert. Zustandsänderungen eines *Models* führen über den *Controller* zu Aktualisierung der GUI (Grafische Benutzerschnittstelle).

4.3.4 Fabrik und Fabrikmethoden

Um die Objekte der Anwendung zu erstellen, wird eine Form einer konkreten Fabrik verwendet [GHJV94]. Diese Klasse hält die Instanzen der Anwendungsobjekte als Komposition. Um an die Referenzen der Instanzen zu kommen, bietet sie Fabrik-Methoden. Die Fabrik muss daher als erste und einzige Instanz erzeugt werden. Dabei übernimmt die Fabrik gekapselt die Initialisierung der *Views* und das Etablieren der MVC-Logik. Die konkrete Implementierung zum Erstellen der Instanzen bleibt in der Fabrik gekapselt und der restlichen Anwendung verborgen. Ein weiterer Einsatz einer Fabrik-Methode ist die Bereitstellung von Instanzen der Übersichtspyramide verschiedener Referenzmodelle. Diese wird gekapselt von der Fabrik genutzt, um der Anwendung die Instanzen von Pyramiden-*Models* bereitzustellen.

4.3.5 XML Persistenz

Das Lesen und Speichern der Anwendungsdaten findet über einen XML-Persistenzrahmen statt. Dieser liest die allgemeine Konfiguration und die definierten Referenzmodelle der Übersichtspyramiden während der Laufzeit ein und erstellt mit diesen Daten vorinitialisierte Instanzen der zugehörigen Klassen. Man spricht von De-Serialisierung. Eine Serialisierung findet beim Schreiben des Konfigurationsobjekts zurück in eine XML-Struktur statt. Dieses Verfahren ermöglicht die einfache Verarbeitung komplexer Datenbestände und Speicherung in ein einfach lesbares Standardformat, ohne den eigentlichen Code zu überfrachten.

4.3.6 Protokollrahmen

Die Anwendung protokolliert ihr Verhalten und ihre Ereignisse mithilfe eines Protokollrahmens, der die Standardausgabe von Java ablöst und entscheidend erweitert. Die Protokollierung findet zur Entwicklungszeit über die Bildschirmausgabe der Konsole oder Entwicklungsumgebung und in einer Protokolldatei statt. Im Auslieferungszustand der Anwendung wird die Protokollierung ausschließlich über die Protokolldatei durchgeführt. Der Protokollrahmen bietet dazu die Möglichkeit, den protokollierten Ereignissen unterschiedliche Gewichtung zu geben.

Generell werden „Debug“-Informationen, allgemeine Informationen und Warnungen ausgegeben. Im Auslieferungszustand wird die „Debug“-Protokollierung deaktiviert. Die Filter für die Ausgabe der Protokollierung und deren Ausgabeziele können über das in Tabelle 4.1 dargestellte Artefakt *logging.properties* konfiguriert werden, ohne den Quellcode des Produkts ändern zu müssen. Das bietet bei Bedarf die Möglichkeit, auch im Auslieferungszustand z.B. die Entwicklermeldungen ausgeben zu lassen.

4.4 Klassendiagramme

4.4.1 Model-View-Controller

Das Klassendiagramm 4.3 zeigt die Implementierung des MVC-Architekturmusters. Konkrete Klassen können das Muster betreffende Verhalten von den dargestellten, abstrakten Klassen erben. Als Mediator stellt der abstrakte *Controller* Methoden zur Registrierung von *Views* und *Models* bereit. Das Observer-Muster implementiert der *Controller* über das *PropertyChangeListener* Interface, welches Java liefert [Ull09]. Änderungen des *Models* werden darüber ereignisgetrieben an die *Views* vermittelt. Ein abstrakter *View* stellt die Methode *modelPropertyChange()* bereit, um mittels des *Controllers* wiederum an ein *Model* Änderungen delegieren zu können. An einem *Controller* können beliebig viele *Models* und *Views* registriert werden.

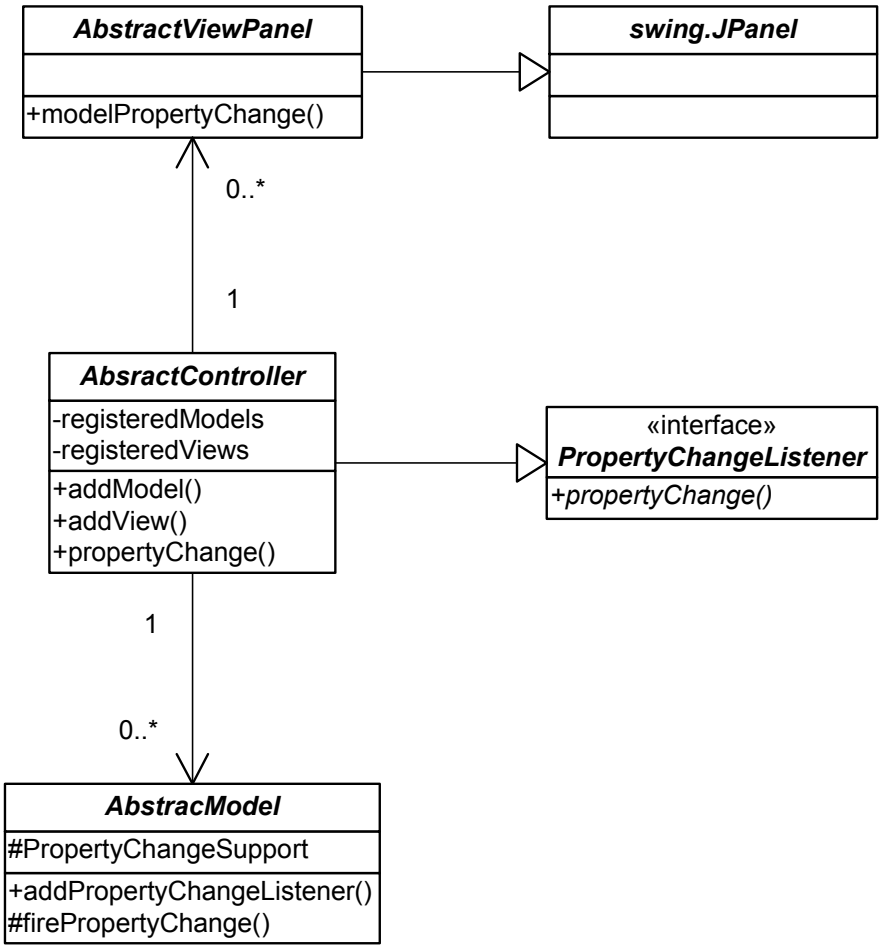


Abbildung 4.3: Klassendiagramm der abstrakten MVC-Zusammenhänge.

4.4.2 Hauptklassen im Kontext der Fabrik

Die Klasse *ModelMeter* ist die von der abstrakten Rhapsody Klasse für Nutzer-Plugins abgeleitete Wurzelinstanz des Plugins und implementiert die Callback API, über welche Rhapsody die Anwendung aufruft. Sie hält die im Kapitel 4.3.4 eingeführte Fabrik des Plugins, welche zur Erzeugung der in der Applikation notwendigen Klasseninstanzen dient. Diese initialisiert die *Views* und etabliert die MVC-Logik. Alle Klassen werden von der Fabrik als Kompositionen gehalten und instantiiert (Abbildung 4.4).

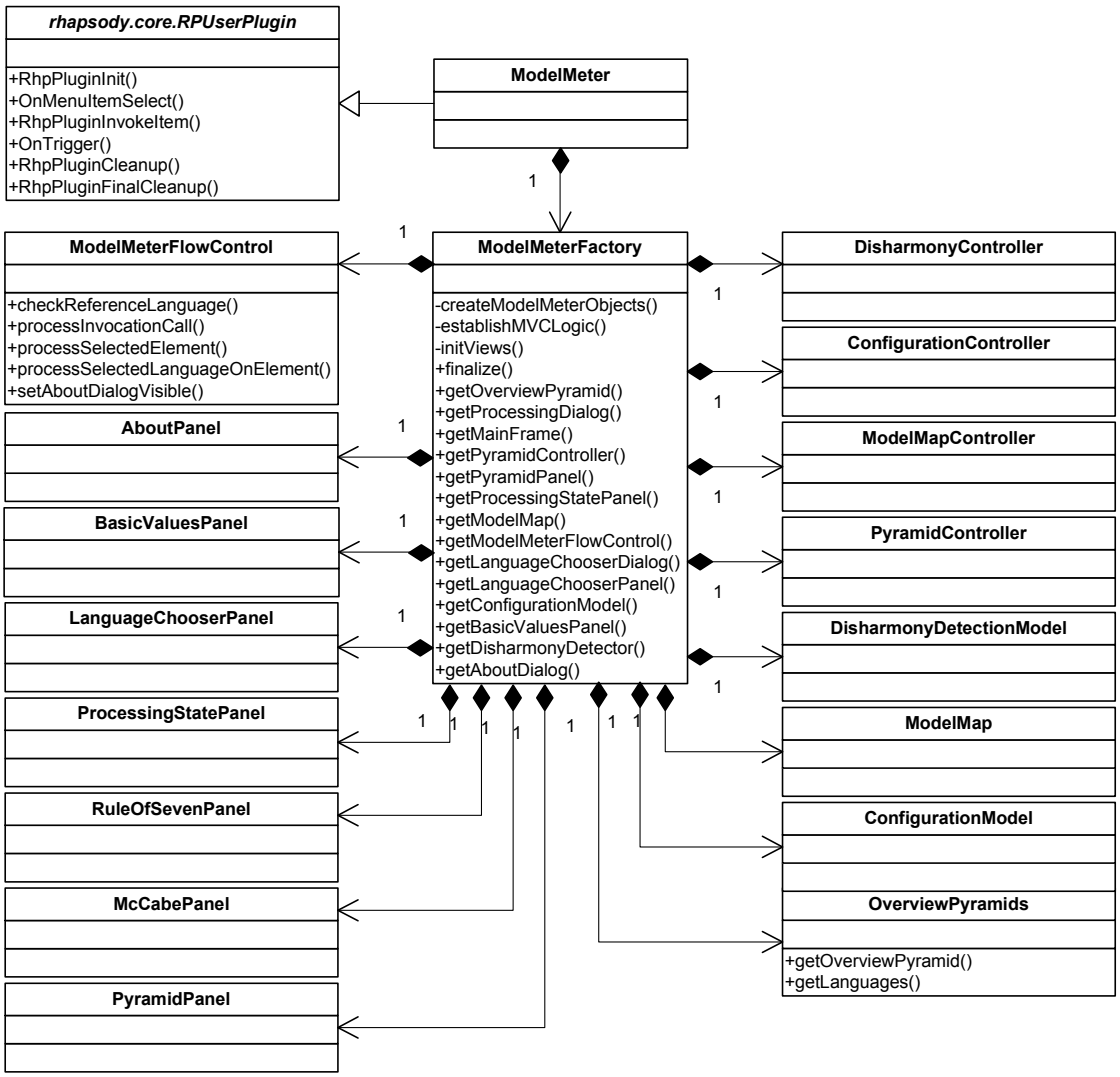


Abbildung 4.4: Klassendiagramm der Fabrikkomposition und Hauptklassen.

Die Fabrik hält zudem alle Referenzen der Dialog Klassen. Diese können in Abbildung 4.4 aber vernachlässigt werden, da sie lediglich als Container für die *Views* dienen.

Die in Abbildung 4.4 gezeigten öffentlichen Methoden der Klasse *ModelMeter* stellen die API-Callback-Methoden für Rhapsody dar. Diese realisieren die Schnittstelle für die Initialisierung, den Aufruf und das Beenden des Plugins.

4.4.3 Die Views

Alle *Views* sind Spezialisierungen des in Abbildung 4.3 gezeigten abstrakten *Views*. Somit können diese am *Controller* registriert und über *Model*-Änderungen informiert werden. Die abstrakte Basis erbt die Java-Klasse *JPanel* der Swing Bibliothek. Diese Grafikbibliothek dient der Anwendung als Programmierschnittstelle zur Darstellung der grafischen Benutzeroberfläche. Sie ist Teil der *Java Foundation Classes* und standardmäßig eine Komponente der Java-Umgebung.

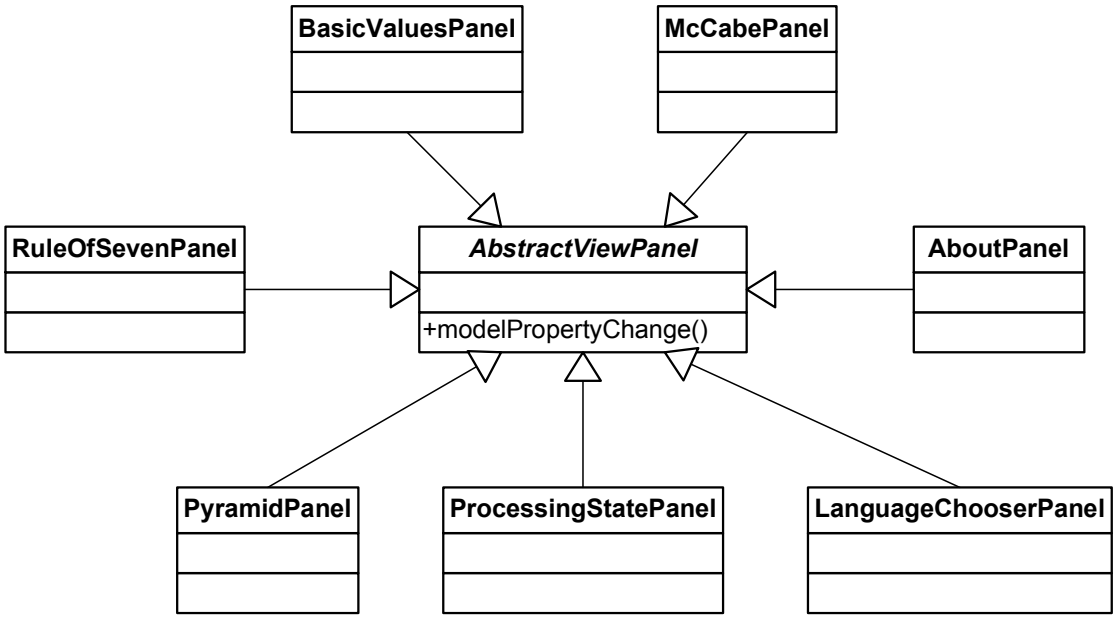


Abbildung 4.5: Klassendiagramm der *Views*.

Wie schon in Kapitel 4.4.2 beschrieben, instantiiert die Anwendung auch Klassen des Typs *JDialog*. Diese dienen aber nur als Container für die *Views* und implementieren darüber hinaus keine Funktionalität. Auch hier werden diese der Einfachheit halber ausgespart. Die einzelnen Funktionen der dargestellten *Views* werden in folgender Tabelle 4.2 beschrieben.

View Panel	Funktion
AboutPanel	Informationen zur Anwendung selbst (Build-Nummer, Autor, ...).
BasicValuesPanel	Direkte Zahlen wichtiger Modellelemente.
LanguageChooserPanel	Auswahlfenster für das Referenzmodell der Übersichtspyramide.
MainView	Hauptfenster der Anwendung mit Werkzeugleiste.
McCabePanel	Darstellung detektierter Zustandsdiagramme bestimmter Komplexität.
ProcessingStatePanel	Fortschrittsfenster zur Darstellung des Verlaufs während der Messung.
PyramidPanel	Darstellung der Übersichtspyramide.
RuleOfSevenPanel	Darstellung detektierter Objektmodelldiagramme bestimmter Komplexität.

Tabelle 4.2: Funktion der *Views*.

4.4.4 Die Controller

Die *Controller* setzen allein die Funktion als Mediator zwischen *Model* und *View* um. Ihre Implementierungsgröße ist naturgemäß sehr gering.

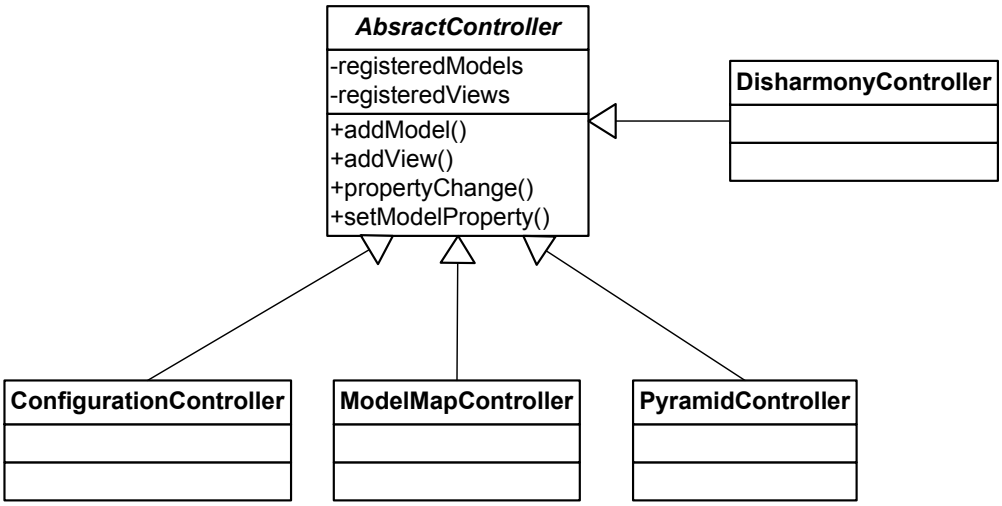


Abbildung 4.6: Klassendiagramm der *Controller*.

Die Anwendung realisiert für jedes *Model* einen *Controller*, über den mehrere *Views* mit *Model*-Änderungen informiert werden. Der eigentliche Kern der Anwendung ist in den *Models* umgesetzt, die im folgenden Kapitel beschrieben werden.

4.4.5 Die Models

Abbildung 4.7 zeigt die vier *Models* der Messanwendung. Sie überschreiben die Methoden ihrer abstrakten *Model*-Basisklasse für die Funktionalität zur Benachrichtigung von Änderungen über die vom *Controller* implementierte *PropertyChangeListener* Schnittstelle.

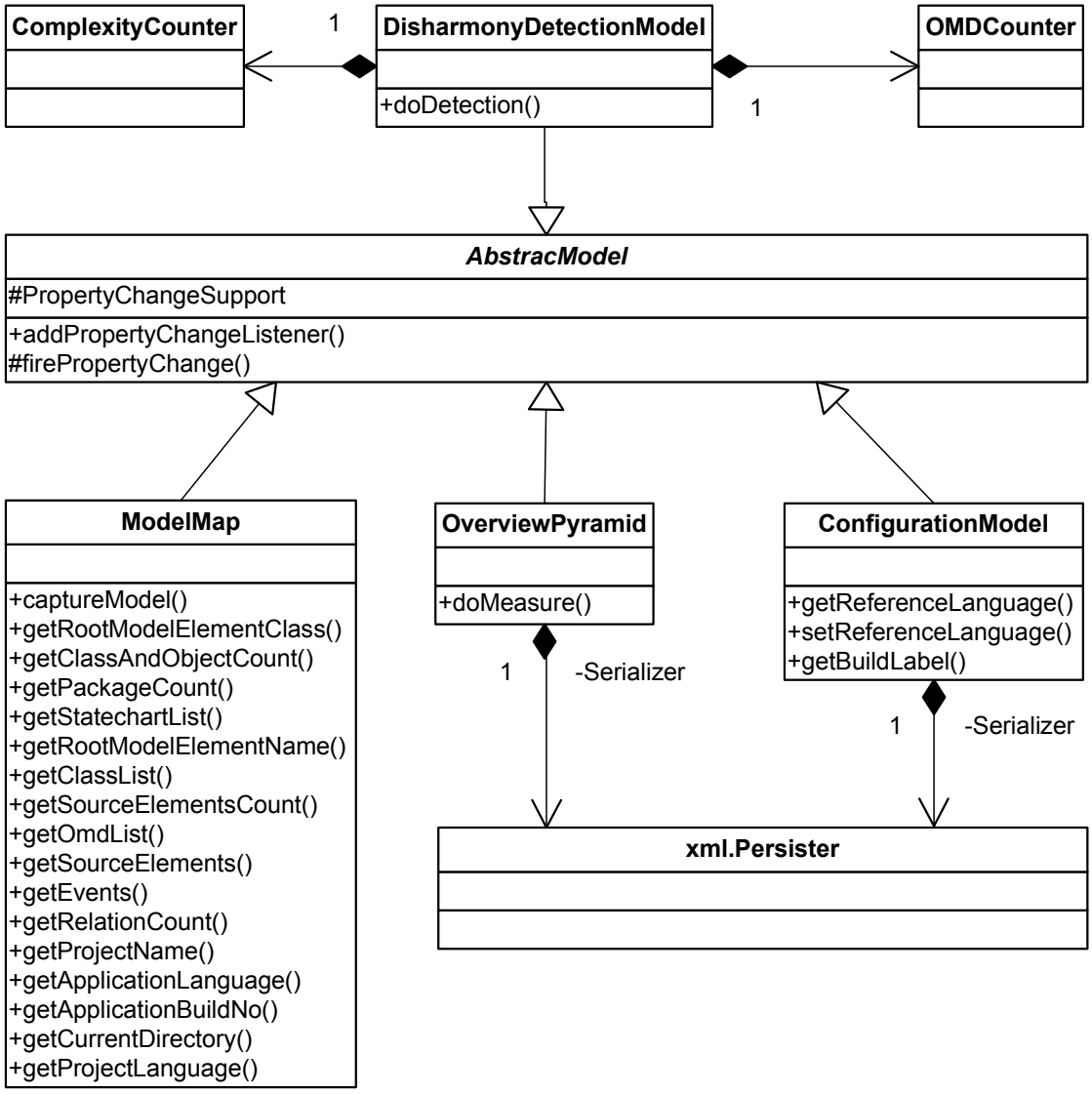


Abbildung 4.7: Klassendiagramm und Zusammenhänge der *Models*.

In Tabelle 4.3 wird die Funktion der *Models* beschrieben.

Model	Funktion
ConfigurationModel	Hält die Konfigurationsdaten der Anwendung (Build-Nummer, aktuell gewähltes Referenzmodell der Übersichtspyramide).
DisharmonyDetectionModel	Realisiert die Funktionalität zur Detektion von Disharmonien.
ModelMap	Die *Model Map* implementiert die Erfassung des zu messenden Rhapsody-Modells und stellt das Abbild dessen relevanter Daten dar. Sie ist zentrale Grundlage für die Messungen und Analyse.
OverviewPyramid	Setzt die Messung und Analyse im Kontext der Übersichtspyramide um. Hält die Daten des Schwellwert-Referenzmodells.

Tabelle 4.3: Funktion der *Models*.

Die *Models* der Übersichtspyramiden und der Konfiguration nutzen die in Kapitel 4.3.5 beschriebe XML-Persistenz Funktionalität. Die Strukturdetails der Übersichtspyramide werden im folgenden Kapitel dargestellt.

4.4.6 Die Übersichtspyramide

Die in Abbildung 4.8 dargestellte Klassenstruktur des *Models* zur Übersichtspyramide zeigt die Klasse *OverviewPyramid*. Diese hält konfigurationsabhängig beliebig viele Pyramiden-Instanzen verschiedener Referenzmodelle bezüglich der Interpretationsschwellwerte. Weiter implementiert sie eine Fabrikmethode, bei der sich die Fabrik mit Referenzen auf eine konkrete Pyramide versorgen kann. Die Auswahl findet durch Übergabe einer Angabe des gewünschten Referenzmodells statt. Die Daten des Referenzmodells liegen wiederum als Aggregation der konkreten Pyramide als einzelne Schwellwertsätze der Metrikkorrelationen vor (*OverviewPyramidThresholdSet*). Zur Messung hält eine Pyramide die verschiedenen Metrikklassen zur Auswertung als Komposition. Die Attribute einer konkreten Pyramide werden bei ihrer Erstellung direkt aus der XML-Konfiguration (*overviewPyramids.xml*) mittels XML-Persistenz-Framework zur Laufzeit de-serialisiert. Die Anzahl und Ausprägung der konkreten Pyramiden kann dadurch beliebig konfiguriert werden.

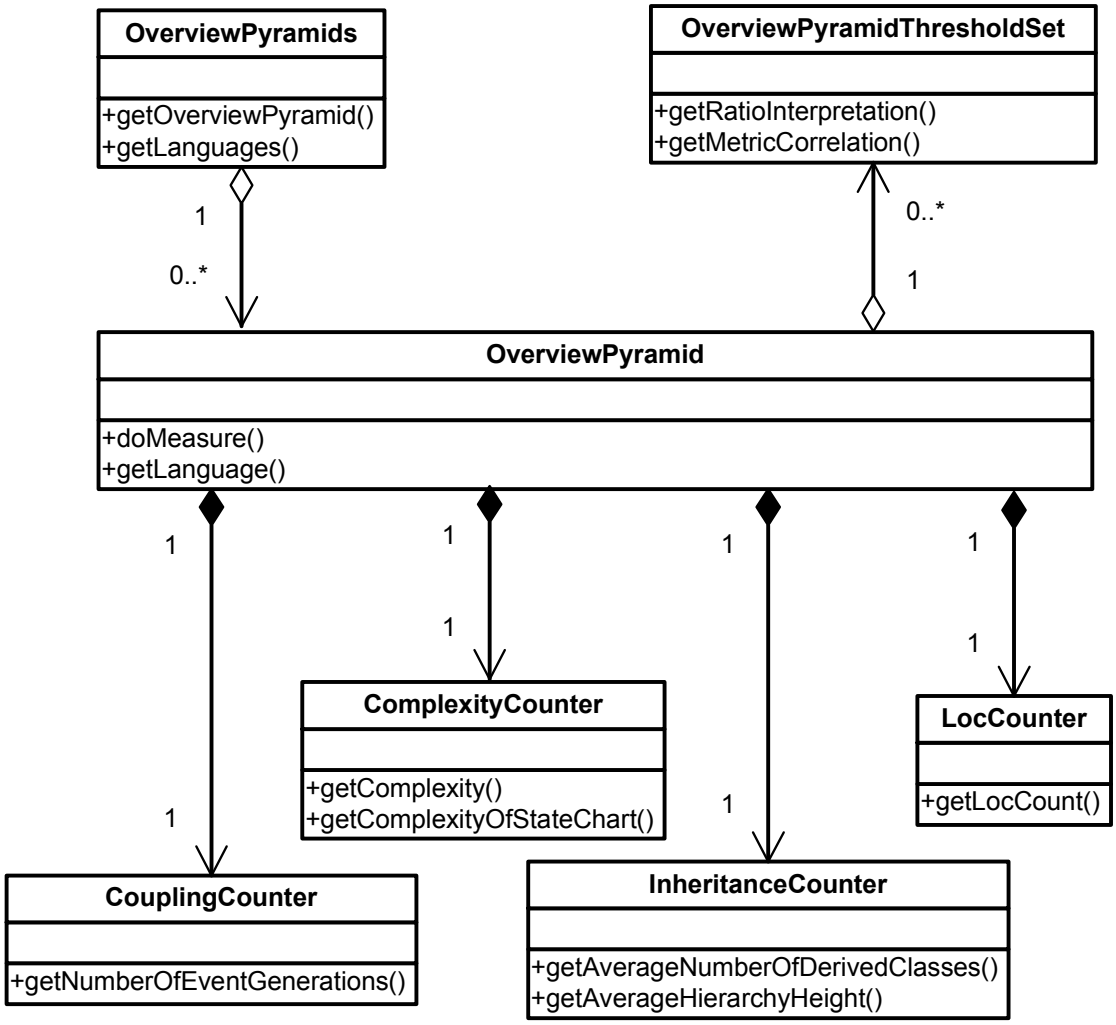


Abbildung 4.8: Klassendiagramm der Übersichtspyramide.

4.5 Verhalten

4.5.1 Die Initialisierung

Die in Kapitel 4.4.2 in Abbildung 4.4 vorgestellten öffentlichen API-Callback-Methoden werden auch bei Initialisierung benutzt (Abbildung 4.9). Konkret ruft Rhapsody beim Öffnen eines Projekts die Methode textitRhpPluginInit() auf. Dabei wird eine Instanz der Plugin-Fabrik erstellt. Unterdessen instantiiert diese die restlichen Objekte der Anwendung, initialisiert die *Views* und etabliert die MVC-Logik. Die Hauptklasse *ModelMeter* holt sich eine Referenz auf die Ablaufsteuerung (*ModelMeterFactory*) von der Fabrik. Die Ablaufsteuerung wird daraufhin aufgerufen, um eine Prüfung durchzuführen, ob bereits ein Referenzmodell ausgewählt ist. Ist dies nicht der Fall, wird der Benutzer aufgefordert, dies zu tun. Damit ist das Plugin initialisiert.

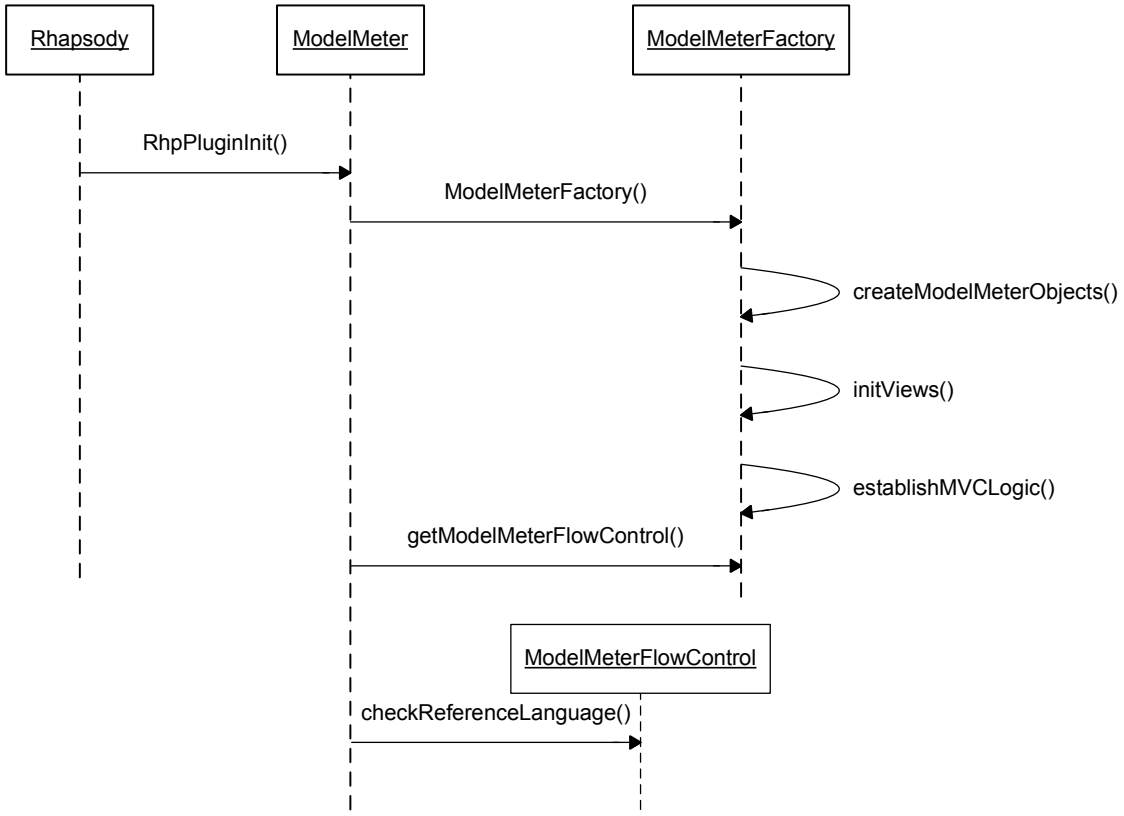


Abbildung 4.9: Initialisierung des Plugins.

4.5.2 Aufruf über das „Tools“ Menü.

Das Werkzeug Rhapsody bietet die Möglichkeit, das Plugin über sein Menü mit dem Namen „Tools“ aufzurufen. Dieser Aufruf ist parameterlos und ruft wiederum die Methode *processInvocationCall* der Ablaufsteuerung auf. Dabei wird das aktive Rhapsody-Projekt mit übergeben. Die Methode führt die Auswertung durch (s. Abbildung 4.13).

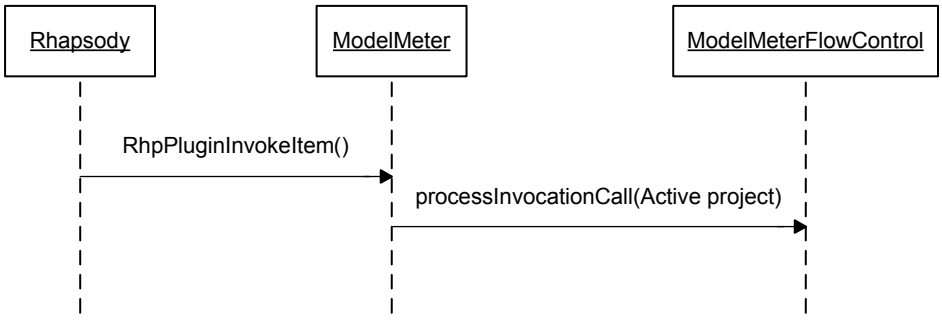


Abbildung 4.10: Aufruf des Plugins aus dem „Tools“ Menü.

4.5.3 Aufruf über das Kontextmenü

Weiter ist es möglich, das Plugin über das Kontextmenü eines Modellelements im Modell-Browser aufzurufen. Dabei wird das selektierte Element übergeben. Wie im vorangegangenen Kapitel beschrieben, löst das den Aufruf der Ablaufsteuerung aus (Abbildung 4.11).

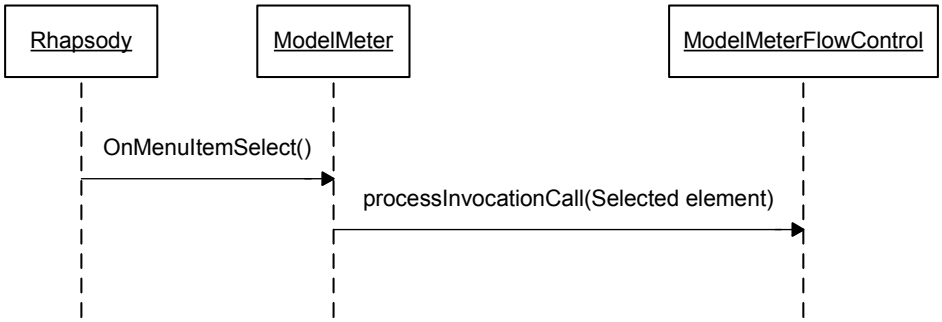


Abbildung 4.11: Aufruf des Plugins aus dem Kontextmenü.

4.5.4 Aufruf aufgrund eines Ereignisses

Rhapsody kann so konfiguriert werden, dass das Plugin beim Auftreten eines bestimmten Ereignisses aufgerufen wird. Bei diesem Ereignis kann es sich um das Öffnen des Projekts, das Sichern des Projekts, dem Zeitpunkt vor einer Generierung des Quellcodes, das Ergänzen eines neuen Modellelements und weitere Ereignisse handeln. Die

Ablaufsteuerung wird hierbei mit dem aktuell selektierten Element zur Messung aufgerufen (Abbildung 4.12)

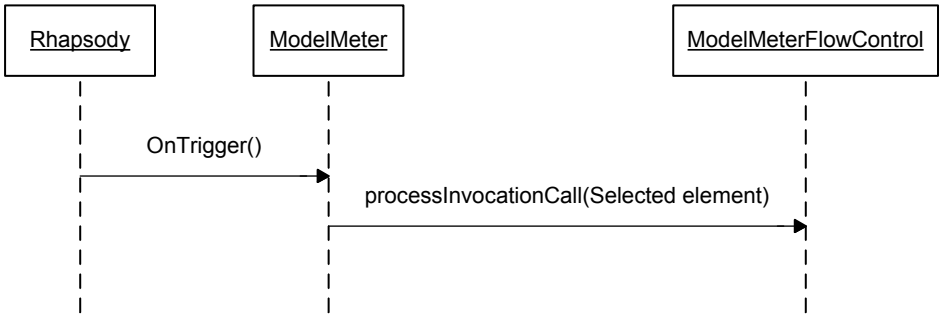


Abbildung 4.12: Aufruf des Plugins aufgrund eines Ereignisses.

4.5.5 Ablauf einer Messung - Ablaufsteuerung

Ob über das Menü oder mittels Ereignis, die Ablaufsteuerung behandelt diese Aufrufe identisch (Abbildung 4.13). Diese holt sich die Konfiguration, die *Model Map* (s. Kapitel 4.4.5), das *Model* zu Detektion von Disharmonien und eine Übersichtspyramide. Die Pyramide ist abhängig von dem in der Konfiguration hinterlegten Metrik-Referenzmodell. Daraufhin wird die *Model Map* über die Methode *captureModel()* mit dem Abbild des übergebenen Modelelementes gefüllt. Dieses Abbild wird wiederum in den Aufrufen zur Pyramidenmessung und Detektion von Disharmonien übergeben.

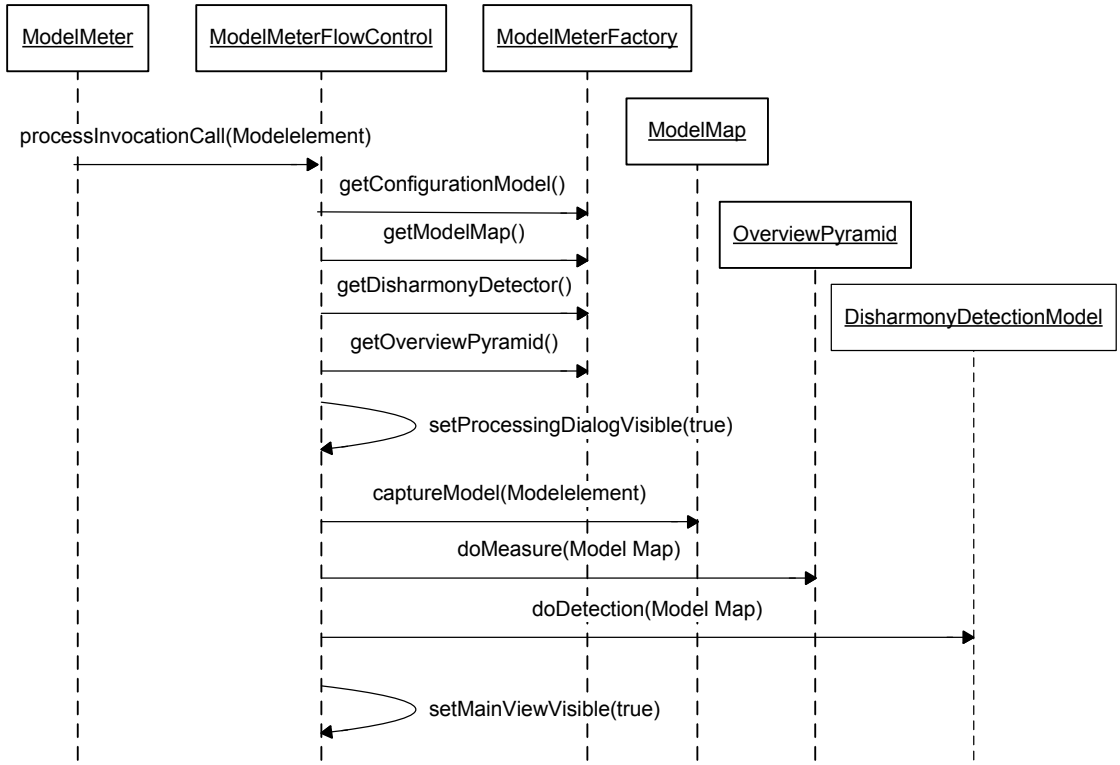


Abbildung 4.13: Ablaufsteuerung nach Aufruf des Plugins.

4.5.6 Ablauf einer Messung - Pyramide

Bei Messung der Pyramidenwerte werden Metrikresultate ermittelt, die Verhältniszahlen der Korrelationen berechnet, die Verhältnisse anhand der Schwellwerte interpretiert und die Ergebnisse protokolliert. Die Aktualisierung der Pyramidenpräsentation geschieht über die MVC-Logik. Bei der Messung der Pyramidenmetriken werden erstens direkte Metriken der *Model Map* genutzt und zweitens die Klassen zur Ermittlung der Komplexität, der Kopplung, Vererbungsqualität und der Anzahl der Quelltextzeilen aufgerufen.

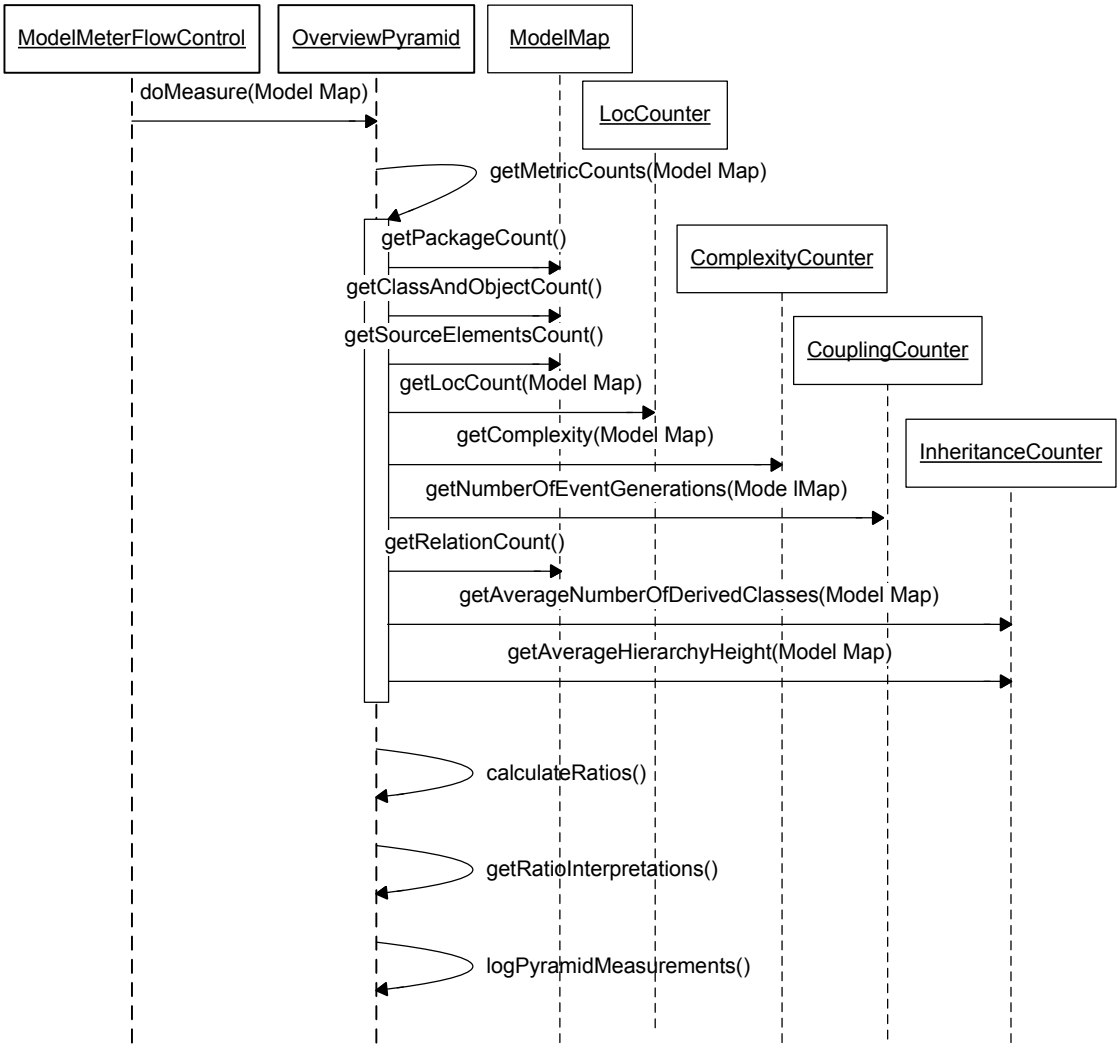


Abbildung 4.14: Ablauf bei der Messung einer Übersichtspyramide.

4.5.7 Ablauf einer Messung - Disharmonien

Die Detektion der Modell-Disharmonien involviert für alle Zustandsautomaten und Objektmodelldiagramme die Methoden zu deren Komplexitätsbestimmung. Alle Elemente, welche dabei über einem vom Nutzer bestimmten Schwellwert liegen, werden auf der Präsentationsebene dargestellt.

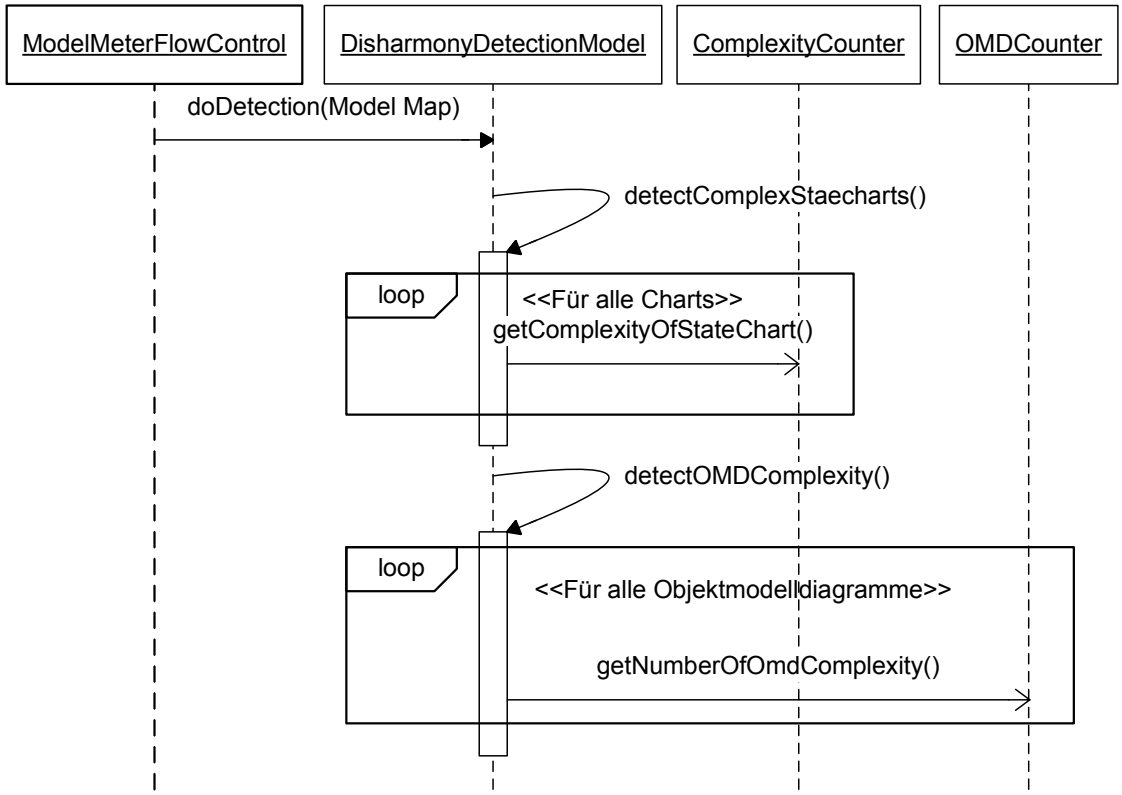


Abbildung 4.15: Ablauf bei der Detektion von Disharmonien.

4.5.8 Beenden der Anwendung

Wird Rhapsody oder das Projekt geschlossen, werden die in Abbildung 4.16 gezeigten Methoden aufgerufen. Die Fabrik wird zur Selbstlöschung veranlasst.

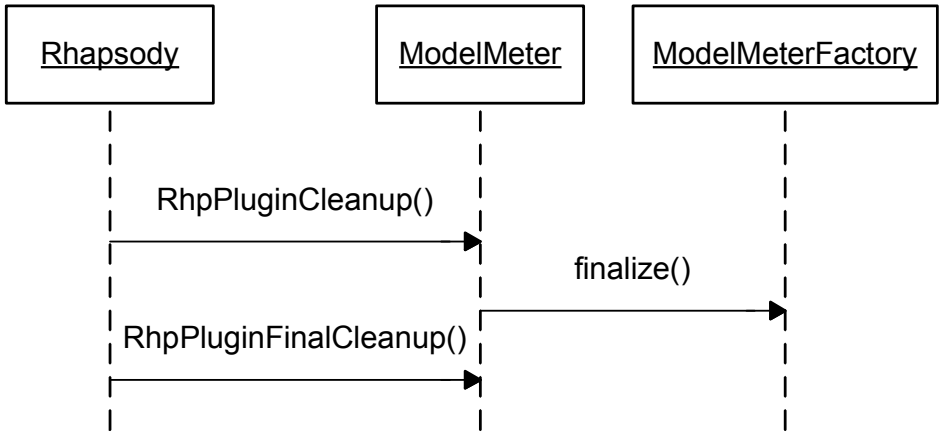


Abbildung 4.16: Beenden des Plugins beim Schließen des Projekts.

4.5.9 Anwenderschnittstelle

Abbildung 4.17 zeigt ein Mockup der Benutzerschnittstelle. Mithilfe einer Baumansicht können die Ansichten zur Darstellung der Basisdaten, der Pyramide und der detektierten Disharmonien zur Darstellung in der Hauptansicht ausgewählt werden. Eine Menüleiste gibt Zugriff auf allgemeine Funktionen der Anwendung wie Messoptionen und Hilfe. Die in Kapitel 4.4.3 beschriebenen *Views* finden hier ihre Darstellung.

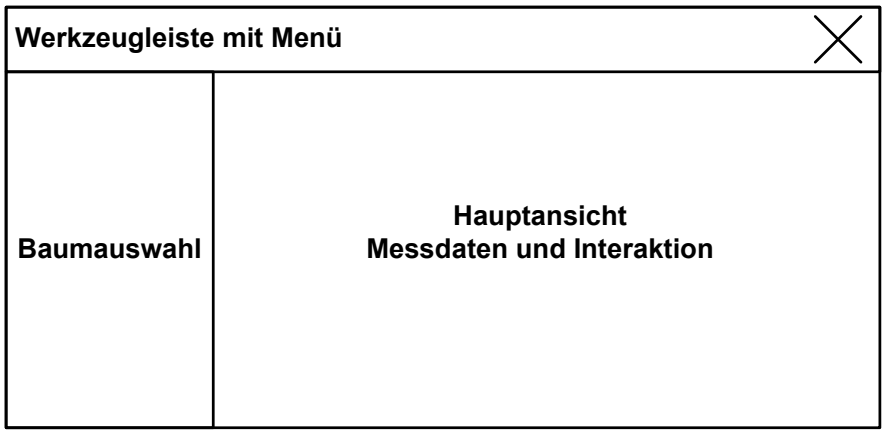


Abbildung 4.17: GUI Mockup der Messanwendung.

Eine detaillierte und vollständige Beschreibung von Funktion, Ansichten und Bedienung der Benutzerschnittstelle ist in der Bedienungsanleitung der Messanwendung in Anhang A zu finden.

4.6 Verwendete Bibliotheken und Quellen

4.6.1 Protokollierungsrahmen

Als Protokollierungsrahmen wird die Bibliothek *log4j* der Apache Software Foundation in Version 1.2.15 verwendet [Apa09]. Diese bietet die Möglichkeit, die in Kapitel 4.3.6 beschriebenen Ereignisse in beliebigen Ausgabeformaten zu protokollieren. Dies können z.B. die Konsole, eine Textdatei, ein System-Protokollierungsdienst und weitere Ausgabemöglichkeiten sein. Weiter können Meldungen unterschiedlich gewichtet werden, was eine feine Granulierung der Ausgabe zulässt. Als Beispiel können Entwicklermeldungen nur zur Entwicklungszeit generiert werden, während Nutzerwarnungen immer per Mail verschickt werden. Der Zusatzaufwand bei der Implementierung hält sich gering, da die verwendeten Protokoll-Objekte zentral von einem einmal erstellten Eltern-Objekt ihre Eigenschaften vererbt bekommen. Einer Klasse ist daher lediglich ein Protokoll-Objekt mitzugeben, um dessen einfache Protokollmethoden nutzen zu können.

4.6.2 Persistenzrahmen

Wie in Kapitel 4.3.5 dargestellt, wird das Lesen und Schreiben komplexer Strukturen mittels XML-Persistenz implementiert. Das geschieht mit dem XML-Persistenz-Framework Simple in Version 2.1.4. [Gal09]. Die Persistenz wird durch XML De-Serialisierung während der Laufzeit umgesetzt. Ein Persistenzobjekt erstellt Klasseninstanzen, deren Attribute so markiert werden können, dass diese aus XML-Dateien initialisiert werden. Die Markierung der Attribute findet mittels *Java Annotationen* in der Klassendeklaration statt.

4.6.3 MVC-Fragmente

Die abstrakten Klassen der MVC-Struktur sind aus dem von Robert Eckstein [Eck07] vorgestellten Verfahren übernommen. Übliche Probleme von ereignisgetriebenen MVC-Strukturen wie Benachrichtigungsschleifen sind von vornherein mittels der Nutzung des Java-eigenen *PropertyChangeSupport* adressiert und ausgeschlossen. Delegiert ein *Controller* Änderungen eines *Views* an ein *Model*, so wird diese Änderung durch die eingangs erwähnte Methode nicht direkt wieder an den *View* gemeldet.

4.6.4 Modultestrahmen

Die im Rahmen der Entwicklung durchgeführten Modultests und deren Regressionsläufe wurden über den Java-Testrahmen *JUnit* in Version 4.7 implementiert [JUn09]. Dieser Testrahmen bietet im wesentlichen eine umfangreiche Sammlung an

Regelmethoden, mit deren Hilfe das Verhalten der Anwendung auf Gültigkeit gestestet werden kann. Diese betriebserprobte Bibliothek bietet ein zuverlässiges Umfeld für verlässliche Tests. Natürlich nur, sofern diese auch korrekt spezifiziert sind.

4.6.5 LOC Zähler Fragmente

Die Klasse zur Zählung der Programmzeilen portiert und refaktorisiert Teile der Werkzeugsammlung *SLOCCount* [Whe09]. Im Speziellen ist das ein ursprünglich in C implementiertes Code-Fragment, das einen Zustandsautomaten zur Auswertung von C, C++ und Java Code realisiert. Dieses wurde während der Portierung an die Zählung der Programmzeilen von Code enthaltenden Rhapsody Modelelementen angepasst.

4.7 Qualitätsmaßnahmen und Werkzeuge

4.7.1 Standard Bibliotheken

Die in Kapitel 4.6 beschriebenen Bibliotheken und Code-Fragmente sind betriebserprobte Open Source Produkte. Sie werden in unzähligen Referenzimplementierungen täglich in verschiedensten Bereichen erfolgreich eingesetzt. Diese Tatsache und die üblich hohe Qualität stark genutzter Open Source Produkte überträgt sich in den genutzten Bereichen auch auf die hier implementierte Software-Anwendung. Die Implementierung der Bibliotheken bleibt gekapselt.

4.7.2 Geringe Abhängigkeiten

Die verwendeten Bibliotheken stehen für sich alleine. Sie benötigen standardmäßig nur die Java-Laufzeitumgebung und keine weiteren externen Bibliotheken. Das verringert die Kopplung des Systems und die Funktionalitäten sind in ihren jeweiligen Einheiten gebunden.

4.7.3 Code Konvention

Als organisatorische Maßnahme werden die Java-Code-Konventionen zur Implementierung des Codes und dessen Kommentare angewandt. Eine Anforderung der Stakeholder ist der Wunsch nach guter Wartbarkeit des Quellcodes. Es ist üblich, dass 80 Prozent der Kosten während des Lebenszyklusses einer Software für Wartung ausgegeben werden [Mic97]. An dieser Stelle erhöht das Einhalten einer etablierten Konvention die Lesbarkeit des Quellcodes und verringert die Einarbeitungsschwelle für weitere Autoren.

Zudem ist nicht nur das Plugin im Auslieferungszustand als das Produkt anzusehen, sondern auch der Quellcode selbst. Dieser sollte mit demselben Qualitätsanspruch behandelt werden.

4.7.4 Geringe Kopplung

Neben der in Kapitel 4.7.1 beschriebenen Bindungs- und Kopplungseigenschaften, wird großer Wert auf die Verringerung negativer Aspekte in diesem Kontext gelegt. Die Verwendung strikter Objektorientierung sowie der Umsetzung angemessener Architektur- und Entwurfsmuster (s. Kapitel 4.3) soll dem Design der Implementierung möglichst positive Kopplungs- und Bindungseigenschaften einbringen.

4.7.5 Statische Analyse

Während der Implementierung wurde als analytisches Verfahren die statische Analyse, mithilfe des Werkzeugs *Checkstyle* in Version 5, durchgeführt [Bur09]. Anfänglich als Prüfer für die Einhaltung von Code Konventionen entwickelt, liefert das Werkzeug aktuell eine große Sammlung an Regeln. Diese gehen inzwischen über die reine Ästhetikprüfung hinaus und schließen auch designrelevante Aspekte mit ein.

4.7.6 Modultests

Als testendes Verfahren werden Modultests für die Prüfung der Metriklassen durchgeführt. Dafür wird der in Kapitel 4.6.4 beschriebene Testrahmen instrumentalisiert. Basierend auf einem für die diese Tests implementierten Rhapsody-Modell, werden Äquivalenzstichproben und deren Grenzen getestet.

Kapitel 5

Experimentelle Messung

5.1 Zugrunde liegende Daten

Um eine Ermittlung geeigneter Schwellwerte für die Interpretation der Proportionen bezüglich der Übersichtspyramide zu ermitteln, ist eine repräsentative Datenbasis essentiell. Aus den zur Verfügung stehenden Modellen werden 32 ausgesucht (Anhang B), welche die Implementierungssprache C++ besitzen. Die Quelle der Modelle ist die Standardinstallation von Rhapsody. Dabei wurden funktionslose Mustermodelle ausgeschlossen. Es handelt sich bei den Projekten um Beispiele und Referenzimplementierungen. „Reale“ Modelle aus industriellem Einsatz konnten zur Ergänzung der Datensammlung nicht herangezogen werden. Da es sich bei den gemessenen Systemen um objektorientierte Embedded-Anwendungen handelt, ist zu erwarten, dass die Paradigmen Klassifikation und Kapselung zwar genutzt werden, dagegen Polymorphismus und Vererbung nur sehr konservativ eingesetzt werden. Die Messreihen der Vererbungsmaße sollten also einen hohen Anteil sehr kleiner Werte aufzeigen.

Grundsätzlich können die Ergebnisse nur so aussagekräftig sein, wie die Qualität der Datenbasis. Daher wird in diesem Kapitel neben der Schwellwertberechnung eine Interpretation der Ergebnisse bezüglich ihrer Repräsentativität durchgeführt. Ohne Beurteilung der Qualität der Ausgangsdaten verlässt man sich im schlimmsten Fall auf unbrauchbare Schwellwerte. Zwar kann eine Validierung einen solchen Missstand aufdecken, bis dahin würde man sich aber auf ungültige Ergebnisse verlassen.

5.2 Messergebnisse

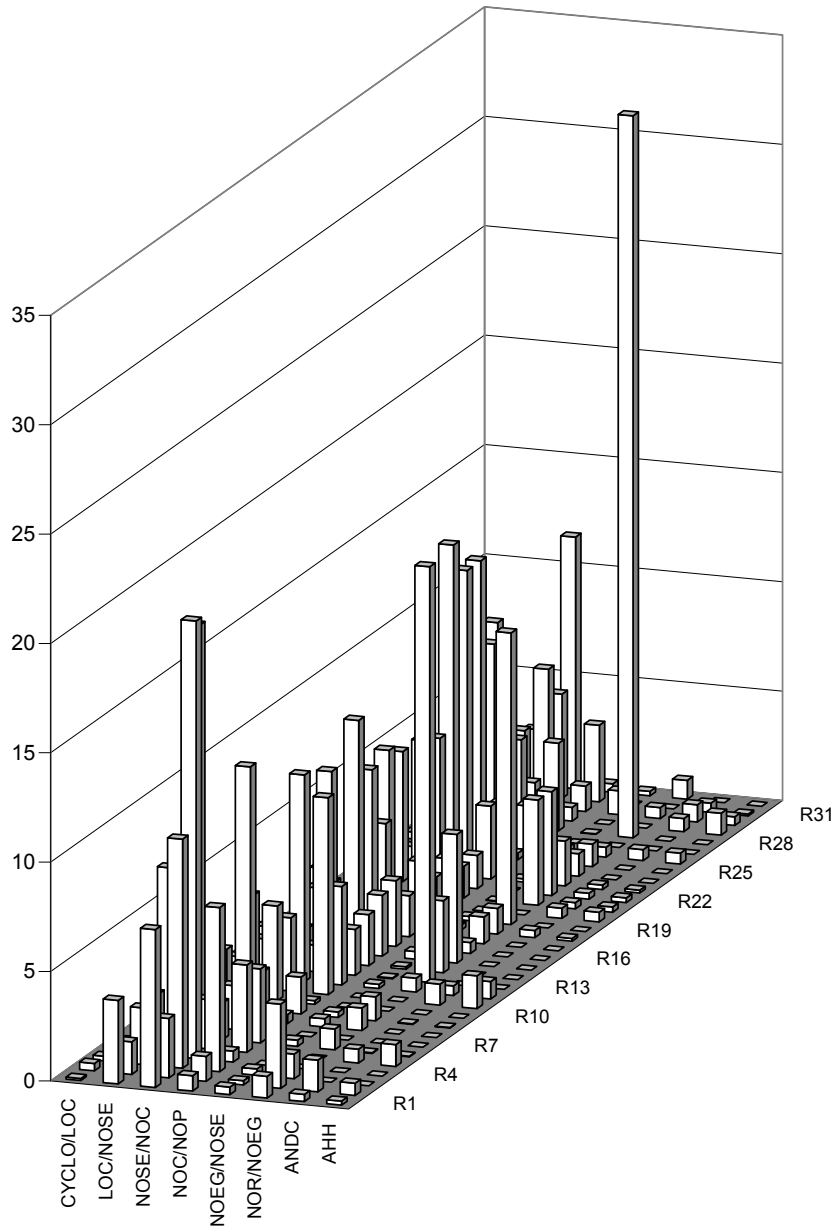


Abbildung 5.1: Proportionsreihen für ausgewählte Projekte in C++.

Die reine Anzahl von 32 Messreihen sollte eine quantitativ ausreichende Grundlage bilden, um eine Aussage über die Qualität der Daten anstellen zu können. Folgend werden zunächst die ermittelten Schwellwerte gelistet. Diese berechnen sich wie folgt [LM06] aus dem arithmetischen Mittel der Proportionsreihe und der Standardabweichung:

Unterer Schwellwert:	Arithmetisches Mittel - Standardabweichung
Durschnittschwellwert:	Arithmetisches Mittel der Proportionsreihe
Oberer Schwellwert:	Arithmetisches Mittel + Standardabweichung

	Low	Average	High
CYCLO/LOC	0,01	0,23	0,55
LOC/NOSE	0,01	3,25	6,59
NOSE/NOC	1,74	6,56	11,37
NOC/NOP	0,37	2,33	4,3
NOEG/NOSE	0,01	0,18	0,4
NOR/NOEG	0,01	3,21	9,87
ANDC	0,01	0,24	0,59
AHH	0,01	0,23	0,59

Tabelle 5.1: Normalisierte Schwellwerte.

Bei der Berechnung der in Tabelle 4.1 gelisteten Schwellwerte zeigte sich auffällig häufig eine größere Standardabweichung zum Mittelwert. Dadurch entstanden bei der Subtraktion teils negative untere Schwellwerte, welche auf den positiven Wert 0,01 normalisiert wurden, um im Ansatz praktisch zu sein. Dieser Umstand sollte hier Indikator sein, um sich skeptisch mit den Messreihen zu beschäftigen.

5.3 Interpretation

Betrachtet man die Proportionsreihen genauer, fällt die starke Abweichung zur gaußschen Normalverteilung auf (Anhang B). Diese Vermutung der Sichtprobe wird durch die Durchführung des Kolmogorow-Smirnow-Tests auf Normalverteilung gestärkt [JL07]. Es ergibt sich für alle Reihen als Ergebnis eine signifikante Abweichung von der Normalverteilung. Das erklärt auch die teils hohen Standardabweichungen, welche zu den sehr niedrigen normalisierten Schwellwerten führen. Gut zu sehen ist das exemplarisch an der Boxplot- und Histogrammdarstellung der LOC/NOSE Proportion in den Diagrammen 5.2 nach [FP96] und 5.3.

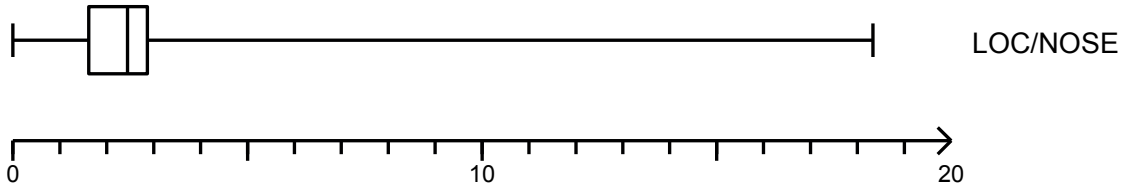


Abbildung 5.2: Box-Whisker-Plot für gemessene Proportionsreihe LOC/NOSE.

Die im Boxplot erkennbare Datenlage spiegelt sich in ähnlicher Form bei den restlichen Proportionen wieder. Das untere Quartil liegt in der Nähe des Nullpunkts. Weiter wiederholt sich ein zur Spannweite auffallend kleiner Quartilsabstand. Das Maximum ist bei fast allen Reihen ein extremer Ausreißer. Sind diese Verteilungen für die vererbungsrelevanten Reihen vielleicht noch ansatzweise erklärbar (s. Kapitel 5.1), ist das für die Gesamtheit nicht mehr möglich. Es muss davon ausgegangen werden, dass etwas nicht stimmt.

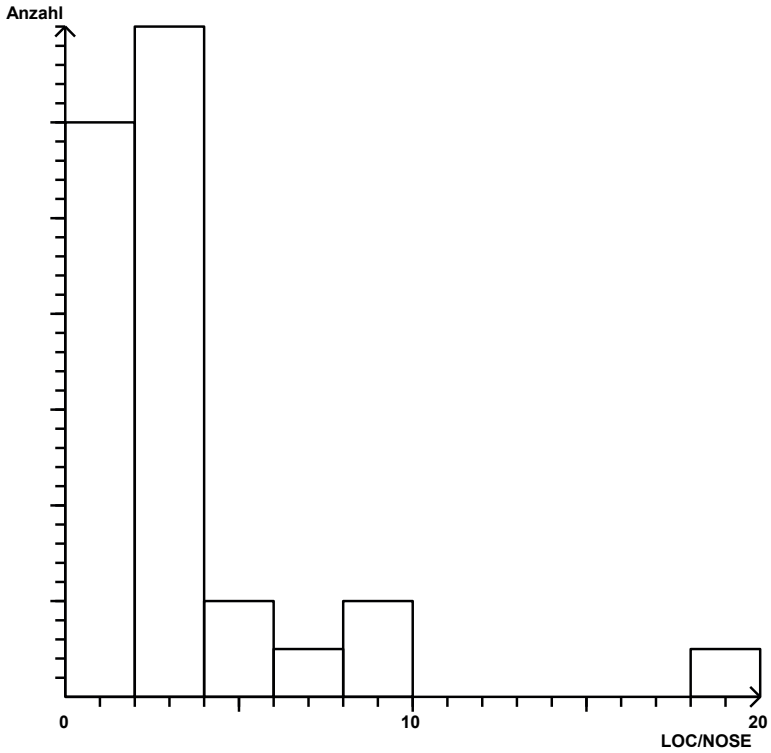


Abbildung 5.3: Histogramm für gemessene Proportionsreihe LOC/NOSE.

Die Abweichung zur Normalverteilung ist in den Histogrammen am deutlichsten sichtbar. Die in Abbildung 5.3 beispielhaft dargestellte Häufigkeitsverteilung ist auch wieder in ähnlicher Form bei den restlichen Proportionsreihen zu finden. Diese durchgängige und große Abweichung macht die Schwierigkeiten bei der Berechnung der Schwellwerte nachvollziehbar.

Es muss davon ausgegangen werden, dass die selektierten Systeme in diesem Kontext nicht zur Datenerhebung geeignet sind. Die kalkulierten Schwellwerte sollten daher als nicht repräsentativ angesehen werden. Die Tatsache, dass die Systeme zwar quantitativ ausreichend sind, aber qualitativ nur Beispielcharakter und Referenzcharakter besitzen, schlägt sich in der Brauchbarkeit der Reihen deutlich nieder. Um an belastbare Schwellwerte zu gelangen, ist die wiederholte Erhebung mit Projekten aus der Industrie unumgänglich.

5.4 Validierung

Eine Validierung der Schwellwerte erübrigt sich aufgrund der im vorangegangenen Kapitel aufgezeigten Unbrauchbarkeit der Eingangsdaten. Sollte es in Zukunft allerdings eine Überprüfung eines neu erhobenen Referenzmodells für die vorliegende Domäne geben, ist dafür eine geeignete Methode zu finden.

Dazu kann die in Kapitel 2.4.2 vorgestellte Goal-Question-Metric-Methode in Teilen angewandt werden. Es könnte dafür das folgende Ziel mit einem GQM-Template zur Zieldefinition beschrieben werden [Gen02] [BW84] [BSL99].

Objekt der Studie	Metriken für IBM Rhapsody Modelle.
Mit der Absicht	Feststellung der Gültigkeit.
Unter Berücksichtigung	Tauglichkeit zur Bewertung der Modellqualität.
Aus der Sicht der Stakeholder	Software Entwickler, Software Architekten, Werkzeugberater.
Kontext	Erstellung einer Messanwendung für IBM Rhapsody.

Tabelle 5.2: Zieldefinition mittels GQM-Template.

Für die Planung ist das Verfahren einer Validierung durch Gegenüberstellung denkbar. Dabei können die ordinal skalierten (s. Kapitel 2.5.2) Pyramideninterpretationen mit äquivalenten Aussagen von Domänenexperten verglichen werden. Das können Entwickler oder Architekten sein, welche eine fundierte Stellungnahme zu den Eigenschaften ihres speziellen Modells treffen können. Die Fragen und Metriken ergeben sich aus dem bereits gewählten Verfahren (s. Kapitel 3). Nach der Sammlung der Daten können Vergleichswerte mithilfe eines Erhebungsbogens aufgenommen werden. Ein Entwurf dafür ist in Anhang C zu finden. Die gesammelten Vergleichswerte können zur statistischen Ermittlung der Gültigkeit des Metrik-Referenzmodells führen.

Kapitel 6

Schlussbetrachtungen

6.1 Ergebnisse der Arbeit

Die Arbeit entwickelte auf Grundlagen der modellgetriebenen Software Entwicklung und Metriken ein Umsetzungskonzept einer Messanwendung für das Werkzeug IBM Rational Rhapsody. Dabei wurde die Wichtigkeit für das Einbeziehen geeigneter Qualitätsmodelle bei der Benutzung von Metriken aufgezeigt. Im Anschluss wurde die für die Messanwendung entworfene Architektur, deren Entwurfsmuster und Verhalten dargestellt. Im Rahmen einer experimentellen Messung zur Ermittlung von Schwellwerten des angewandten Modells zeigte sich die große Bedeutung der Auswahl der Datengrundlage. Diese stellte sich bei deren Analyse als nicht ausreichend repräsentativ heraus. Eine Validierung wurde daher nur als Methode vorgestellt und konnte nicht sinnvoll durchgeführt werden.

6.2 Diskussion und Ausblick

Die umgesetzten Verfahren bieten grundsätzlich das Potential, die Anforderungen an eine praxistaugliche Messanwendung zu erfüllen. Unumgänglich ist aber die Datenerhebung und Berechnung von Schwellwerten für die angewandte Übersichtspyramide mit Daten aus der Industrie. Liegen diese vor, kann eine Validierung deren Gültigkeit zeigen und die Messanwendung bedenkenlos zur Bewertung eingesetzt werden.

Das vorgestellte Verfahren ist ein erster Schritt in Richtung der vereinfachten Darstellung von Messergebnissen. Mithilfe weiterer Visualisierung von Metriken, wie TreeMaps oder polymetrischen Ansichten, bieten sich umfangreiche Möglichkeiten, um Entwicklern das Verständnis ihrer Modelle zu erleichtern. Ein weiteres interessantes Verfahren ist die animierte Darstellung der Evolution eines Modells. Neben der Visualisierung könnte die Methode der Detektionsstrategien für Anti-Pattern und *Bad Smells* erweitert und mit statistischer Erhebung gesichert werden [LM06]. Dadurch könnten gezielt Problemzonen betrachtet und gegebenenfalls refaktorisiert werden.

Diese Arbeit warf einen technischen Blick auf das Thema Metriken im Kontext der MDSD. Es darf bei aller Faszination für die Möglichkeiten aber nicht vergessen werden, dass bei der Umsetzung und Einführung dieser oder ähnlicher Verfahren großes Fingerspitzengefühl auf der sozialen und der Prozessebene gefordert ist. Metriken sollen unterstützen, aufklären und helfen. Allzu schnell werden deren Ergebnisse fehlinterpretiert oder im falschen Kontext benutzt. Das heißt, die Verfahren müssen offen und methodisch sauber in den jeweiligen Entwicklungsprozess eingebettet werden. Dann bieten Metriken eine sinnvolle Ergänzung auf dem Weg zu besserer Software und Modellen.

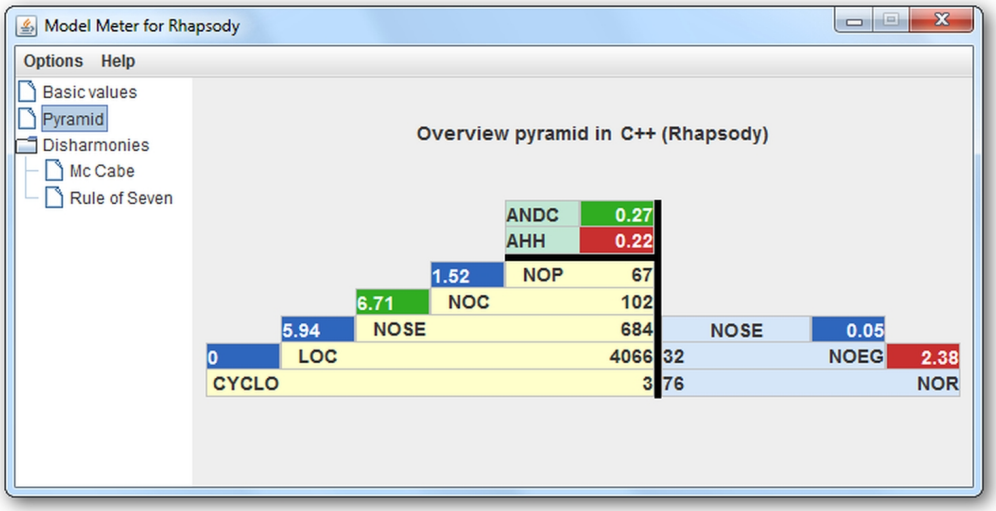
Anhang A

Model Meter Manual

Model Meter for Rhapsody Manual

IBM Rational Software Germany
Georg Simon Ohm University Nuremberg

Thomas Goertz - modelmeter@tomgo.de – 2009



Contents

Introduction

Model Meter for Rhapsody is a Java Plugin for measuring basic object oriented aspects of your Rhapsody model. It provides an overview for getting a first impression of an existing model or attends continous quality feedback while development.

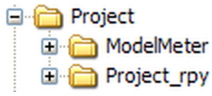
Note: For more detailed infomation about language reference models, configuration, architecture aspect, etc. please refer to the appropriate master thesis.

Technical Requirements

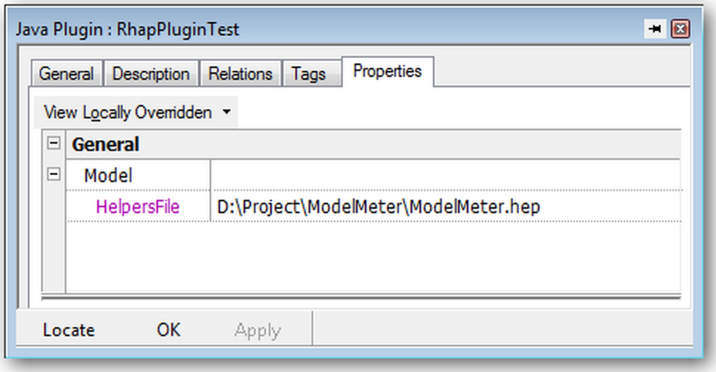
- IBM Rhapsody 7.5
- Java Runtime Environment 1.6

Installation

1 Unzip the Model Meter Package contents into the root folder of your project you want to measure like this:

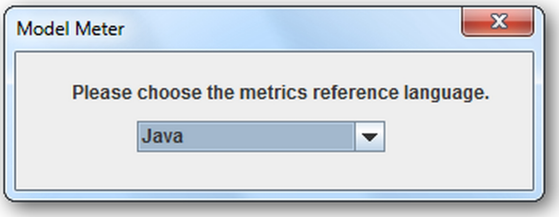


2 Open the project and choose the ModelMeter.hep inside the ModelMeter folder as your HelpersFile as seen like here:



3 Save and restart Rhapsody

4 Now you are asked to choose your reference language. Choose what fits your model`s language best:

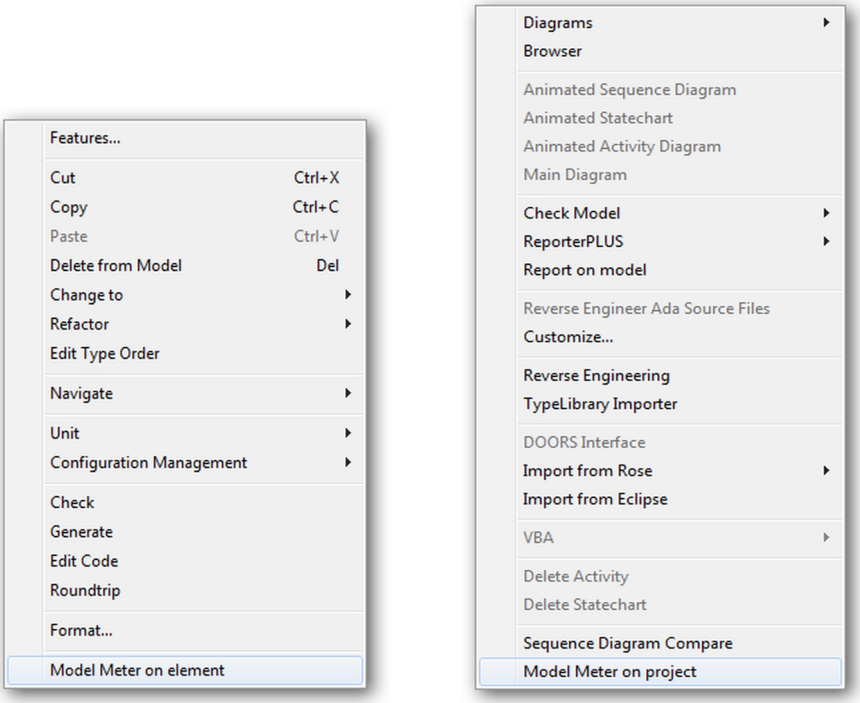


Using Model Meter

After the Plugin is installed, you can choose it from

1 the bottom of the tools menu and measure your whole project model.

2 context menu of any model element from the model browser you want to measure.



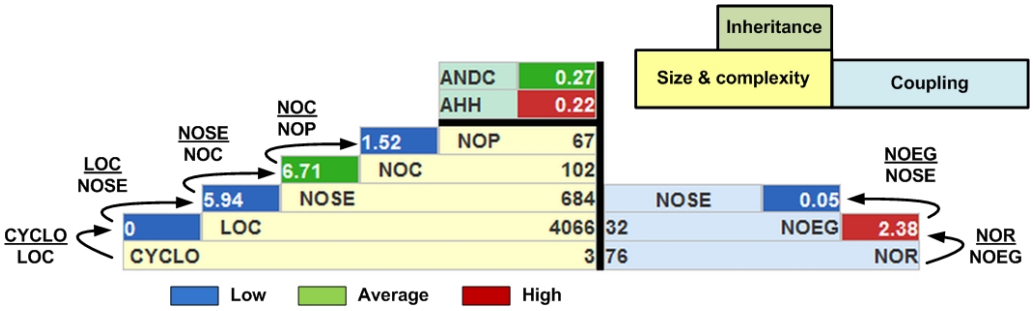
After the measurements are taken, you can choose from the following tree view`s options of the main window:

- Basic Values
- Overview Pyramid
- Mc Cabe
- Rule of Seven

The functionality of each option is described below.

Overview Pyramid and its interpretation

The Overview Pyramid gives a display to get a first impression of the measured model. It shows direct metric values and their calculated proportions, separated in three regions, concerning the aspect of size & complexity, inheritance and coupling. Based on the thresholds of the chosen reference language model, the proportions are scaled in low, average and high values, marked with colors.



The following metrics are taken:

- ANDC — Average number of derived classes
- AHH — Average hierarchy height of derived classes

- NOP — Number of packages
- NOC — Number of classes
- NOSE — Number of source elemets
- LOC — Lines of code
- CYCLO — McCabe's cyclomatic complexity

- NOEG — Number of event generations
- NOR — Number of relations

Interpretation of the size ratios

From the NOC/NOP ratio down to the CYCLO/LOC ratio the proportions can be read as a ladder into the size aspects of the model.

NOC/NOP This proportion gives an impression if the packages tend to be coarse or fine grained on a high level structure view.

NOSE/NOC This proportion delivers a first picture of the class design's quality. It shows how source elements are distributed among classes. High values could figure out missing classes or unrefactored reactive behaviour.

LOC/NOSE High values of this proportions could be an indiactor for more "big" source elements. You can get a first indentation how the source is distributed over the source elements.

CYCLO/LOC This poroportion gives you an impression of how the statechart or activitychart`s complexity correlates to the source`s size. High values could be a hint for a more conservative use of the reactive tools options.

Interpretation of the coupling ratios

NOEG/NOSE This proportion indicates the collaboration between the objects in the event driven context. Very high values could be a signal for excessive coupling on this architecture level. A lot of events are sent from a lot of source elements.

NOEG/NOR This porportion is a hint, of how much event driven coupling involves many classes.

Interpretation of the inheritance values

The inheritance metric values are not claculated as proportions.

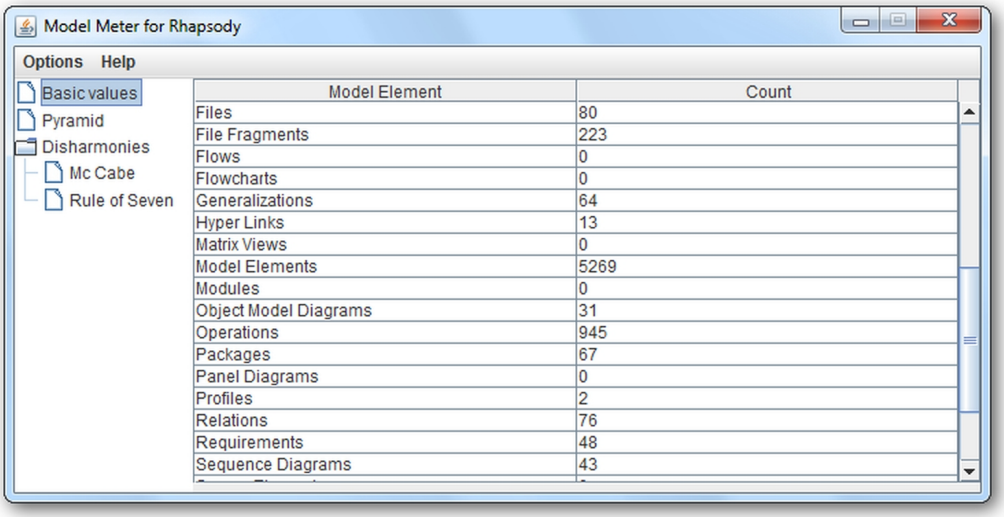
ANDC The average number of derived classes gives an impression of how inheritance "width" is.

AHH The average hierarchy height of derived classes gives an impression of how inheritance "height" is.

These two metrics show you how intense the use of inheritance is.

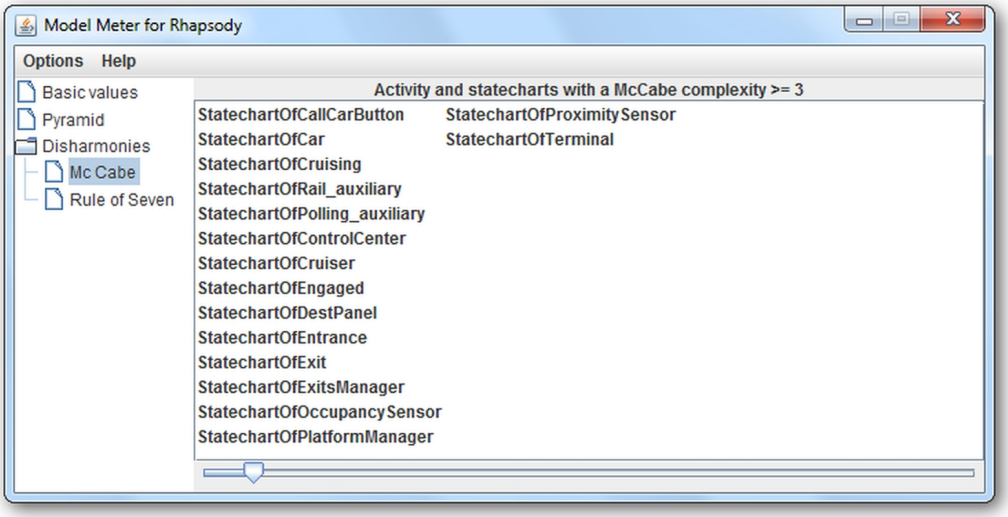
Basic Values

The basic value panel provides a list of interesting direct element counts.



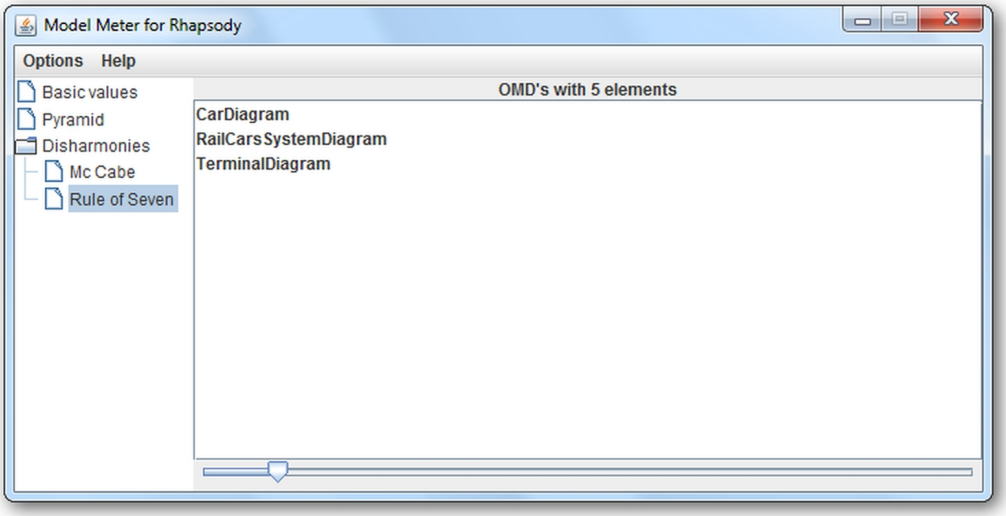
Mc Cabe

The Mc Cabe panel shows the statecharts and activity diagrams of your project model with a complexity greater than the chosen one. Adjust the complexity via the slider at the bottom of the window for needs. By clicking a list element it will show up in the rhapsody browser.



Rule of Seven

The Rule of Seven panel shows the object model diagrams with an element count greater than the chosen one. Ajust the complexity via the slider at the bottom of the window for your needs. By clicking a list element, it will show up in the rhapsody browser. The regarded elements are classes, objects, modules and files.



Menu options

Options menu

Select reference language	*Lets you reselect your reference language model*
Refresh measurment	*Conducts a new measurement of the selected element*
Close	*Closes Model Meter*

Help menu

Model Meter Help	*Shows this manual in your browser if available*
About Model Meter	*Basic information abot the author and the context*

End-User License Agreement

This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity), the end-user, and Thomas Görtz, the author of Model Meter for Rhapsody ("Author"). This EULA permits you to use a single copy, or multiple copies of the software product identified above, which includes computer software and may include associated media, printed materials, and on-line or electronic documentation ("SOFTWARE PRODUCT").

By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bounded by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE PRODUCT.

No Warranties. The Author expressly disclaims any warranty for the SOFTWARE PRODUCT. The SOFTWARE PRODUCT and any related documentation is provided "AS IS" without warranty of any kind, either express or implied, including, without limitation, the implied warranties or merchantability, fitness for a particular purpose, or noninfringement. The entire risk arising out of use or performance of the SOFTWARE PRODUCT remains with you.

No Liability for Damages. In no event shall the Author be liable for any special, consequential, incidental or indirect damages whatever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use this product, even if the Author is aware of the possibility of such damages and known defects.

Anhang B

Measurement Results

Measurement results for the overview pyramid

Measured projects

Date of measurement : 09.11.2009
Project language : C++
Rhapsody language : C++
Project count : 32

from the location ..\IBM\Rational\Rhapsody\7.5\

ADMS.rpy	..\Samples\CppSamples\Atg\Adms4Atg
ADMS.rpy	..\Samples\SystemSamples\Adms
cars.rpy	..\Samples\CppSamples\Cars
CDPlayer.rpy	..\Samples\CppSamples\CD_player
client_SDM_Observers.rpy	..\Samples\CppSamples\CORBA\client_Sdm_observers
CppCashRegister.rpy	..\Samples\CppSamples\TestConductor\CppCashRegister
CppListUsage.rpy	..\Samples\CppSamples\TestConductor\CppListUsage
CppPBX.rpy	..\Samples\CppSamples\TestConductor\CppPbx
CppTestConductorAPI.rpy	..\Samples\CppSamples\TestConductor\CppTestConductorAPI
CppTestingExternalFiles.rpy	..\Samples\CppSamples\TestConductor\CppTestingExternalFiles
CppWebComponent.rpy	..\Share\LangCpp\WebComponents\model
Dishwasher.rpy	..\Samples\CppSamples\Dishwasher
DoDAF_Tutorial.rpy	..\DoDAF Pack\Tutorial\DoDAF_Tutorial_Model
elevator.rpy	..\Samples\CSamples\Elevator
handset.rpy	..\Samples\CppSamples\Handset
hhs.rpy	..\Samples\CppSamples\hhs
HomeAlarm.rpy	..\Samples\CppSamples\HomeAlarm
HomeAlarmWithPorts.rpy	..\Samples\CppSamples\HomeAlarmWithPorts
Mobile.rpy	..\Samples\CppSamples\UML 2.0
NetCentricWeatherService.rpy	..\Samples\SystemSamples\NetCentric\NetCentricWeatherService
oxf.rpy	..\Atg\LangCppOXF\oxf\model
oxf.rpy	..\Share\LangCpp\oxf\model
pacemaker.rpy	..\Samples\CppSamples\Pacemaker
PBX.rpy	..\Samples\CppSamples\Pbx
PBX_atg.rpy	..\Samples\CppSamples\Atg\Pbx4Atg
pingpong.rpy	..\Samples\CppSamples\PingPong
PowerWindowWithSimulink.rpy	..\Samples\CppSamples\PowerWindowWithSimulink
Radio.rpy	..\Samples\CppSamples\Radio
Sdm_Observers.rpy	..\Samples\CppSamples\CORBA\Sdm_Observers
Tests.rpy	..\Share\LangCpp\Validation\oxf\CoreAPI\Models
Tetris.rpy	..\Samples\CppSamples\Tetris
TheVendingMachine.rpy	..\Samples\CppSamples\Atg\TheVendingMachineStart

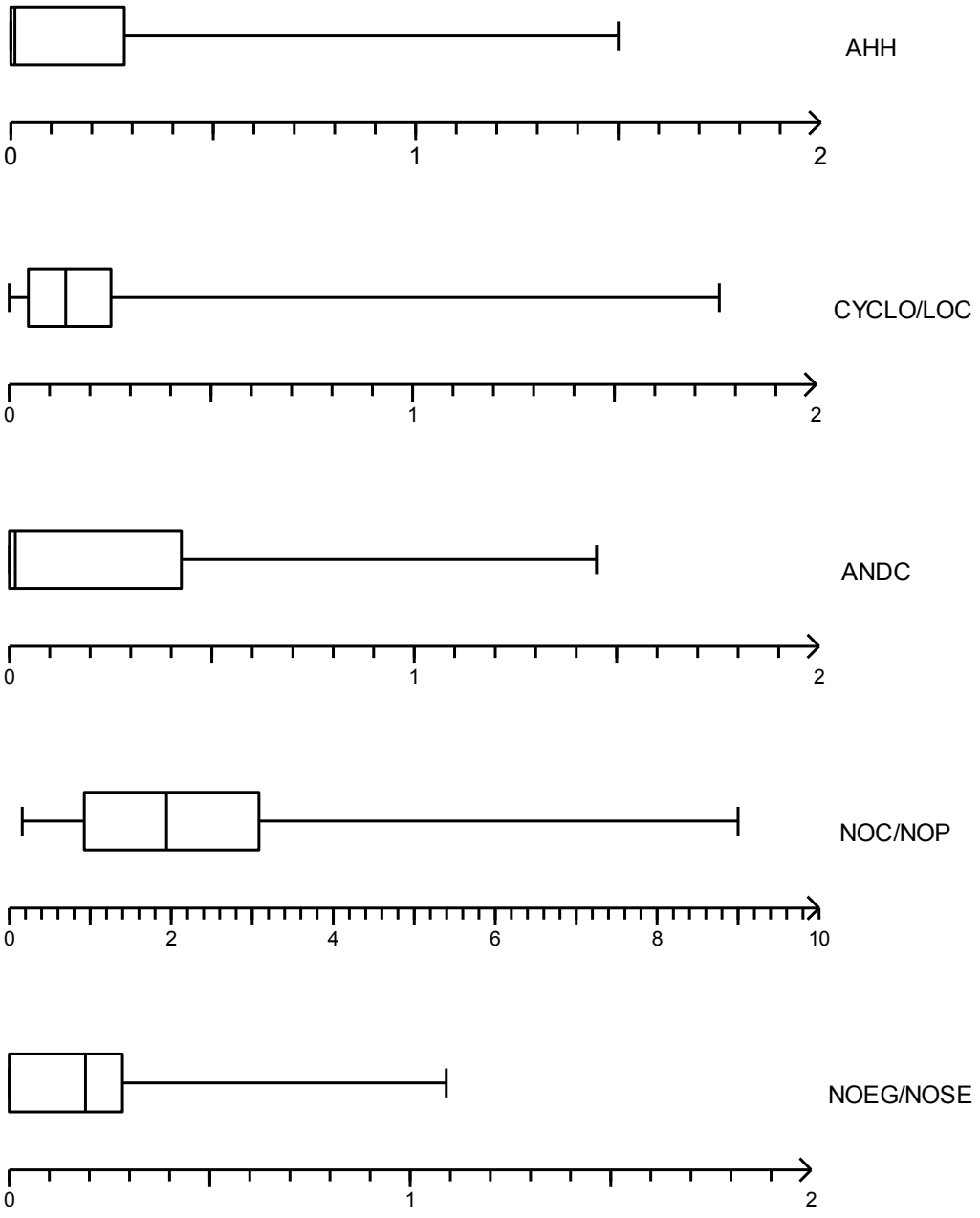
Direct metric results

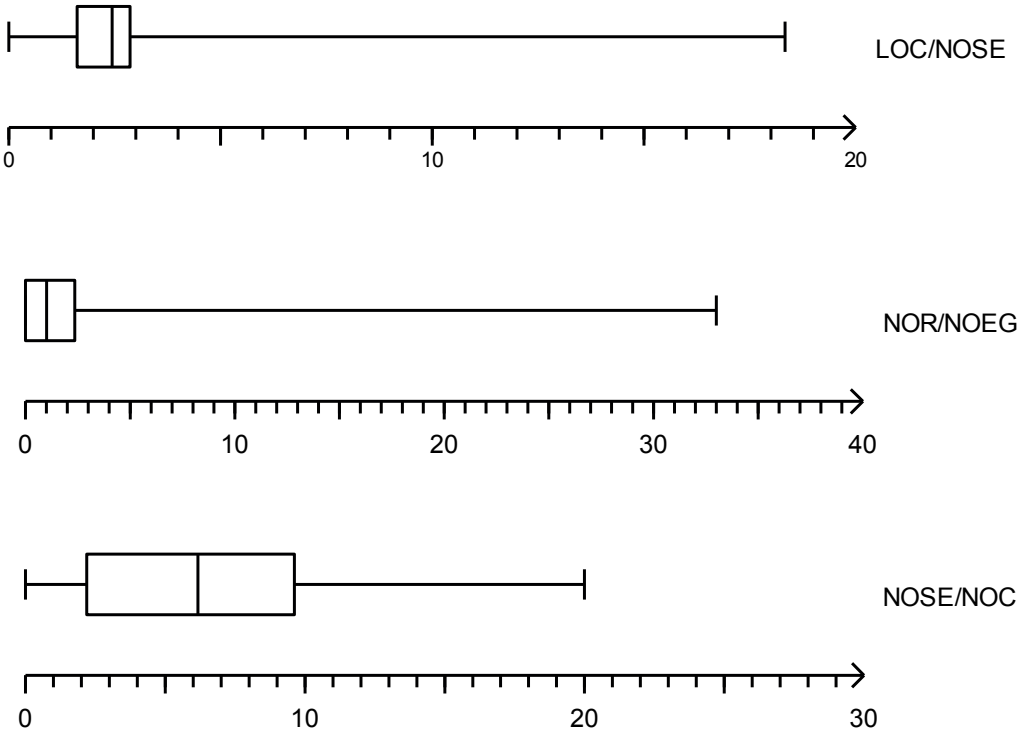
NOP	NOC	NOSE	LOC	CYCLO	NOEG	NOR	ANDC	AHH
83	58	419	1599	149	142	140	0,33	0,18
22	25	68	101	34	13	50	1,45	0,56
2	15	157	407	84	45	51	0	0
2	1	20	56	0	0	0	0	0
2	8	18	146	8	0	2	0,63	1
13	44	74	210	22	21	20	0,03	0,03
17	3	6	110	0	0	3	0	0
17	7	81	222	31	31	32	0	0
28	47	84	221	27	21	23	0,03	0,02
19	3	13	24	0	0	2	0	0
6	54	181	766	0	0	11	0,96	1,5
2	9	85	151	35	16	10	0,44	0,8
20	42	51	53	20	2	38	0	0
3	7	61	101	20	7	23	0	0
9	25	23	22	19	8	47	0	0
2	6	61	143	31	16	8	0	0
8	15	112	244	41	22	27	0	0
7	21	96	137	36	22	26	0,33	0,13
14	26	16	17	30	3	40	0	0
21	47	0	0	0	0	1	0,47	0,44
67	98	690	4442	3	16	77	0,28	0,24
67	102	684	4066	3	16	76	0,27	0,22
3	10	151	174	73	22	45	0,2	0,13
2	6	81	222	31	31	32	0	0
20	6	81	206	31	31	32	0	0
2	4	37	68	22	11	5	0,5	0,5
3	3	18	18	9	0	9	0	0
4	16	64	135	14	1	33	0	0
8	5	8	70	3	0	2	0,6	1
38	44	279	619	157	305	154	0,79	0,4
6	21	100	293	24	0	17	0,42	0,08
16	6	69	175	18	14	12	0	0

Ratio series

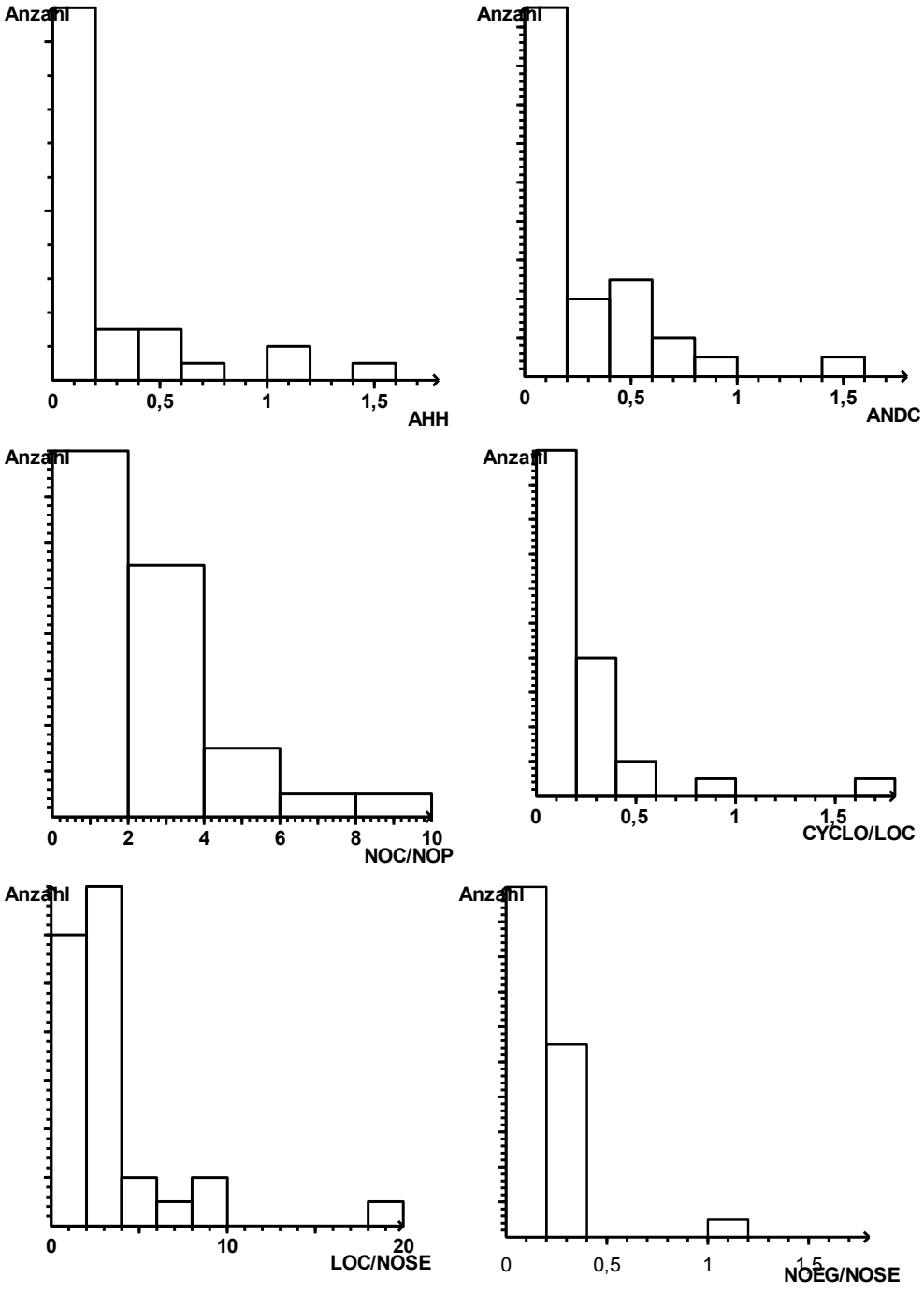
CYCLO/LOC	LOC/NOSE	NOSE/NOC	NOC/NOP	NOEG/NOSE	NOR/NOEG	ANDC	AHH
0,09	3,82	7,22	0,7	0,34	0,99	0,33	0,18
0,34	1,49	2,72	1,14	0,19	3,85	1,45	0,56
0,21	2,59	10,47	7,5	0,29	1,13	0	0
0	2,8	20	0,5	0	0	0	0
0,05	8,11	2,25	4	0	0	0,63	1
0,1	2,84	1,68	3,38	0,28	0,95	0,03	0,03
0	18,33	2	0,18	0	0	0	0
0,14	2,74	11,57	0,41	0,38	1,03	0	0
0,12	2,63	1,79	1,68	0,25	1,1	0,03	0,02
0	1,85	4,33	0,16	0	0	0	0
0	4,23	3,35	9	0	0	0,96	1,5
0,23	1,78	9,44	4,5	0,19	0,63	0,44	0,8
0,38	1,04	1,21	2,1	0,04	19	0	0
0,2	1,66	8,71	2,33	0,11	3,29	0	0
0,86	0,96	0,92	2,78	0,35	5,88	0	0
0,22	2,34	10,17	3	0,26	0,5	0	0
0,17	2,18	7,47	1,88	0,2	1,23	0	0
0,26	1,43	4,57	3	0,23	1,18	0,33	0,13
1,76	1,06	0,62	1,86	0,19	13,33	0	0
0	0	0	2,24	0	0	0,47	0,44
0	6,44	7,04	1,46	0,02	4,81	0,28	0,24
0	5,94	6,71	1,52	0,02	4,75	0,27	0,22
0,42	1,15	15,1	3,33	0,15	2,05	0,2	0,13
0,14	2,74	13,5	3	0,38	1,03	0	0
0,15	2,54	13,5	0,3	0,38	1,03	0	0
0,32	1,84	9,25	2	0,3	0,45	0,5	0,5
0,5	1	6	1	0	0	0	0
0,1	2,11	4	4	0,02	33	0	0
0,04	8,75	1,6	0,63	0	0	0,6	1
0,25	2,22	6,34	1,16	1,09	0,5	0,79	0,4
0,08	2,93	4,76	3,5	0	0	0,42	0,08
0,1	2,54	11,5	0,38	0,2	0,86	0	0

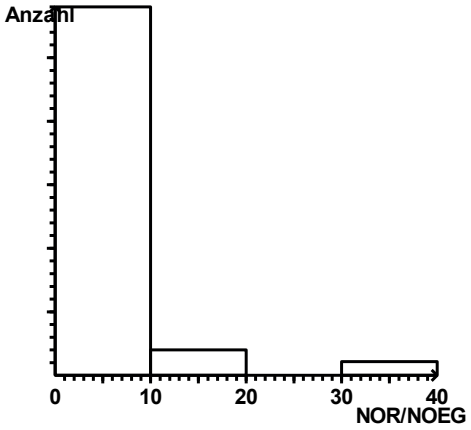
Boxplots for ratio series

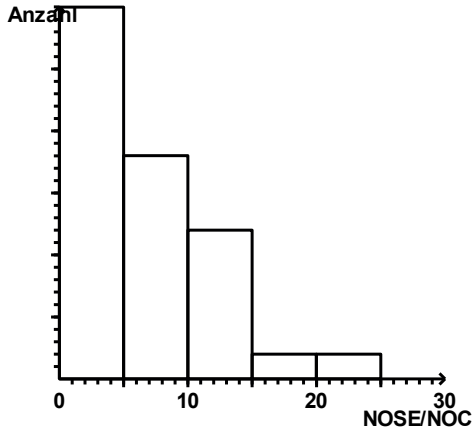



LOC/NOSE
0
10
20
NOR/NOEG
0
10
20
30
40
NOSE/NOC
0
10
20
30

Histograms for ratio series







Thresholds

	STDEV	LOW	AVG	HIGH
CYCLO/LOC	0,33	-0,1	0,23	0,55
LOC/NOSE	3,34	-0,09	3,25	6,59
NOSE/NOC	4,82	1,74	6,56	11,37
NOC/NOP	1,96	0,37	2,33	4,3
NOEG/NOSE	0,21	-0,03	0,18	0,4
NOR/NOEG	6,66	-3,46	3,21	9,87
ANDC	0,34	-0,1	0,24	0,59
AHH	0,37	-0,14	0,23	0,59

Normalized Thresholds

	LOW	AVG	HIGH
CYCLO/LOC	0,01	0,23	0,55
LOC/NOSE	0,01	3,25	6,59
NOSE/NOC	1,74	6,56	11,37
NOC/NOP	0,37	2,33	4,3
NOEG/NOSE	0,01	0,18	0,4
NOR/NOEG	0,01	3,21	9,87
ANDC	0,01	0,24	0,59
AHH	0,01	0,23	0,59

Anhang C

Model Meter Validation Interview Sheet

Model Meter Validation Interview Sheet

Project : _____

Project language : _____

Reference model : _____

Date of measure : _____

Rhapsody build : _____

Model meter build : _____

Survey

Please decide which of the rating (low, average, high) fits your model best for the following questions.

Size & complexity

Do you think the package structuring design granularity is coarse (low) or fine (high)?

NOC/NOP	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Do you think the class structuring design granularity is coarse (low) or fine (high)?

NOSE/NOC	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Do you think the source elements are more small ones (low) or tending to be big (high)?

LOC/NOSE	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Do you think the reactive options (statecharts/activitydiagrams) are used more conservative (low) or more progressive (high)?

CYCLO/LOC	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Coupling

Do you think event driven collaboration is more deliberate (low) or excessive (high)?

NOEG/NOSE	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Do you think event driven class coupling is more deliberate (low) or excessive (high)?

NOEG/NOR	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Inheritance

Do you think inheritance "width" is more narrow (low) or general (high)?

ANDC	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Do you think inheritance "height" is more humble (low) or heavy (high)?

AHH	Expert's opinion	Model Meter's Interpretation
low	☐	☐
average	☐	☐
high	☐	☐

Anhang D

Build Instructions

Build instructions for the Model Meter Eclipse project.

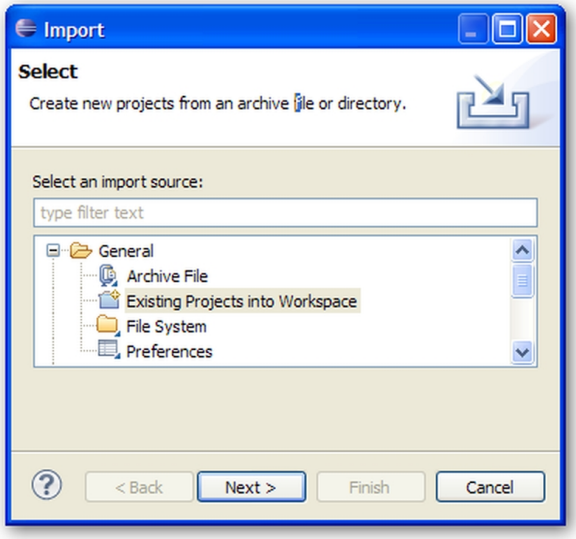
Note: For the environment resources please refer to the Model Meter CD.

Build

1. Install the JRE/JDK 1.6 and Eclipse Galileo
   Optional: Install the visual editor eclipse plugin version 0.9.12
   Optional: Install the fatjar eclipse plugin version 0.0.31

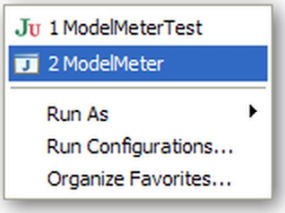
2. **Copy the folder ModelMeter** from the CD's folder Source **into your eclipse workspace folder**.

3. Add the project via the menu **File->Import->Existing Project into Workspace**.



Run

Generate the binaries by making a Clean Build via the menu **Project->Clean...**. For executing the Model Meter, a Rhaspody project needs to be opened. Then choose run Model Meter.



The unit tests are built for running them with the Project RhapPluginTest from the CD.

Anhang E

Model Meter API Documentation

Package
controller

controller
Class AbstractController

```
java.lang.Object
   |
   +-controller.AbstractController
```

All Implemented Interfaces:
java.beans.PropertyChangeListener

Direct Known Subclasses:
ConfigurationController, DisharmonyController, ModelMapController, PyramidController

---

public abstract class **AbstractController**
extends Object
implements java.beans.PropertyChangeListener

This class provides base level functionality for each controller. This includes the ability to register multiple models and views, propogating model change events to each of the views and providing a utility function to broadcast model property changes when necessary.

Constructors

AbstractController

```
public AbstractController()
```

Creates a new instance of Controller.

Methods

addModel

```
public void addModel(AbstractModel model)
```

Binds a model to this controller. Once added, the controller will listen for all model property changes and propogates them on to registered views. In addition, it is also responsible for resetting the model properties when a view changes state.

Parameters:
`model` - The model to be added

---

removeModel

```
public void removeModel(AbstractModel model)
```

Unbinds a model from this controller.

Parameters:
`model` - The model to be removed

---

addView

```
public void addView(AbstractViewPanel view)
```

(continued on next page)

(continued from last page)

Binds a view to this controller. The controller will propogate all model property changes to each view for consideration.

Parameters:

`view` - The view to be added

removeView

```
public void removeView(AbstractViewPanel view)
```

Unbinds a view from this controller.

Parameters:

`view` - The view to be removed

propertyChange

```
public void propertyChange(java.beans.PropertyChangeEvent evt)
```

This method is used to implement the PropertyChangeListener interface. Any model changes will be sent to this controller by using this method.

Parameters:

`evt` - An object that describes the model's property change.

setModelProperty

```
protected void setModelProperty(String propertyName,
          Object newValue)
```

This method fires off property changes back to the models. This method will use reflection to inspect each of the model classes to determine if it is the owner of the property in question. If it isn't, a NoSuchMethodException is thrown.

Parameters:

`propertyName` - The name of the property

`newValue` - An object that represents the new value of the property.

controller
Class ConfigurationController

```
java.lang.Object
   |
   +-
      |
      +-controller.ConfigurationController
```

All Implemented Interfaces:
java.beans.PropertyChangeListener

---

public class **ConfigurationController**
extends AbstractController

MVC Controller for the Configuration functionality of the Model Meter. Acts as a mediator between the registered models and the views.

Fields

CONFIGURATION_BUILDLABEL_PROPERTY

```
public static final java.lang.String CONFIGURATION_BUILDLABEL_PROPERTY
```

Constant value: **`BuildLabel`**

CONFIGURATION_REFERENCELANGUAGE_PROPERTY

```
public static final java.lang.String CONFIGURATION_REFERENCELANGUAGE_PROPERTY
```

Constant value: **`ReferenceLanguage`**

Constructors

ConfigurationController

```
public ConfigurationController()
```

Methods

changeElementReferenceLanguage

```
public void changeElementReferenceLanguage(String referenceLanguage)
```

Changes the reference language property of the configuration model.

Parameters:
`referenceLanguage` - the reference language model which defines the overview pyramids thresholds

controller
Class DisharmonyController

```
java.lang.Object
   |
   +-
      |
      +-controller.DisharmonyController
```

All Implemented Interfaces:
java.beans.PropertyChangeListener

---

public class **DisharmonyController**
extends AbstractController

MVC Controller for the Disharmony Detection functionality of the Model Meter. Acts as a mediator between the registered models and the views.

Fields

DISHARMONY_COMPLEXOMDS_PROPERTY

```
public static final java.lang.String DISHARMONY_COMPLEXOMDS_PROPERTY
```

Constant value: **ComplexOmds**

DISHARMONY_COMPLEXSTATECHART_PROPERTY

```
public static final java.lang.String DISHARMONY_COMPLEXSTATECHART_PROPERTY
```

Constant value: **ComplexStatechart**

DISHARMONY_MCCABETHRESHOLD_PROPERTY

```
public static final java.lang.String DISHARMONY_MCCABETHRESHOLD_PROPERTY
```

Constant value: **McCabeThreshold**

DISHARMONY_OMDTHRESHOLD_PROPERTY

```
public static final java.lang.String DISHARMONY_OMDTHRESHOLD_PROPERTY
```

Constant value: **OmdThreshold**

Constructors

DisharmonyController

```
public DisharmonyController()
```

Methods

changeMcCabeThreshold

```
public void changeMcCabeThreshold(Integer mcCabeThreshold)
```

Changes the Mc Cabe threshold property of the configuration model.

Parameters:

`mcCabeThreshold` - the complexity threshold with that you can detect statecharts and activity diagrams

---

changeOmdThreshold

```
public void changeOmdThreshold(Integer omdThreshold)
```

Changes the Omd Threshold property of the configuration model.

Parameters:

`omdThreshold` - the complexity threshold with that you can detect object model diagrams

controller
Class ModelMapController

```
java.lang.Object
   |
   +-
      |
      +-controller.ModelMapController
```

All Implemented Interfaces:
java.beans.PropertyChangeListener

---

public class **ModelMapController**
extends AbstractController

MVC Controller for the Model Map Model functionality of the Model Meter. Acts as a mediator between the registered models and the views.

Fields

MODELMAP_ACTIVITYDIAGRAMCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_ACTIVITYDIAGRAMCOUNT_PROPERTY
```

Constant value: **activityDiagramCount**

MODELMAP_ACTORCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_ACTORCOUNT_PROPERTY
```

Constant value: **actorCount**

MODELMAP_ANNOTATIONCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_ANNOTATIONCOUNT_PROPERTY
```

Constant value: **annotationCount**

MODELMAP_ATTRIBUTECOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_ATTRIBUTECOUNT_PROPERTY
```

Constant value: **attributeCount**

MODELMAP_CLASSCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_CLASSCOUNT_PROPERTY
```

Constant value: **classCount**

(continued from last page)

MODELMAP_COLLABORATIONDIAGRAMCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_COLLABORATIONDIAGRAMCOUNT_PROPERTY
```

Constant value: **collaborationDiagramCount**

MODELMAP_COMMENTCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_COMMENTCOUNT_PROPERTY
```

Constant value: **commentCount**

MODELMAP_COMPONENTCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_COMPONENTCOUNT_PROPERTY
```

Constant value: **componentCount**

MODELMAP_COMPONENTDIAGRAMCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_COMPONENTDIAGRAMCOUNT_PROPERTY
```

Constant value: **componentDiagramCount**

MODELMAP_CONFIGURATIONCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_CONFIGURATIONCOUNT_PROPERTY
```

Constant value: **configurationCount**

MODELMAP_CONSTRAINTCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_CONSTRAINTCOUNT_PROPERTY
```

Constant value: **constraintCount**

MODELMAP_CONTROLLEDFILECOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_CONTROLLEDFILECOUNT_PROPERTY
```

Constant value: **controlledFileCount**

MODELMAP_DEPENDENCYCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_DEPENDENCYCOUNT_PROPERTY
```

Constant value: **dependencyCount**

MODELMAP_DEPLOYMENTDIAGRAMCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_DEPLOYMENTDIAGRAMCOUNT_PROPERTY
```

(continued from last page)

Constant value: **deploymentDiagramCount**

MODELMAP_DIAGRAMCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_DIAGRAMCOUNT_PROPERTY**

Constant value: **diagramCount**

MODELMAP_EVENTCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_EVENTCOUNT_PROPERTY**

Constant value: **eventCount**

MODELMAP_EVENTRECEPTIONCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_EVENTRECEPTIONCOUNT_PROPERTY**

Constant value: **eventReceptionCount**

MODELMAP_FILECOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_FILECOUNT_PROPERTY**

Constant value: **fileCount**

MODELMAP_FILEFRAGMENTCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_FILEFRAGMENTCOUNT_PROPERTY**

Constant value: **fileFragmentCount**

MODELMAP_FLOWCHARTCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_FLOWCHARTCOUNT_PROPERTY**

Constant value: **flowchartCount**

MODELMAP_FLOWCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_FLOWCOUNT_PROPERTY**

Constant value: **flowCount**

MODELMAP_GENERALIZATIONCOUNT_PROPERTY

`public static final java.lang.String` **MODELMAP_GENERALIZATIONCOUNT_PROPERTY**

Constant value: **generalizationCount**

MODELMAP_HYPERLINKCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_HYPERLINKCOUNT_PROPERTY
```

Constant value: **hyperLinkCount**

MODELMAP_MATRIXVIEWCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_MATRIXVIEWCOUNT_PROPERTY
```

Constant value: **matrixViewCount**

MODELMAP_MODELELEMENTCLASSNAME_PROPERTY

```
public static final java.lang.String MODELMAP_MODELELEMENTCLASSNAME_PROPERTY
```

Constant value: **ModelElementClassName**

MODELMAP_MODELELEMENTCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_MODELELEMENTCOUNT_PROPERTY
```

Constant value: **ModelElementCount**

MODELMAP_MODELELEMENTINDEX_PROPERTY

```
public static final java.lang.String MODELMAP_MODELELEMENTINDEX_PROPERTY
```

Constant value: **ModelElementIndex**

MODELMAP_MODULECOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_MODULECOUNT_PROPERTY
```

Constant value: **moduleCount**

MODELMAP_OBJECTMODELDIAGRAMCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_OBJECTMODELDIAGRAMCOUNT_PROPERTY
```

Constant value: **objectModelDiagramCount**

MODELMAP_OPERATIONCOUNT_PROPERTY

```
public static final java.lang.String MODELMAP_OPERATIONCOUNT_PROPERTY
```

Constant value: **operationCount**

(continued from last page)

MODELMAP_PACKAGECOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_PACKAGECOUNT_PROPERTY`**

Constant value: **`packageCount`**

MODELMAP_PANELDIAGRAMCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_PANELDIAGRAMCOUNT_PROPERTY`**

Constant value: **`panelDiagramCount`**

MODELMAP_PROFILECOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_PROFILECOUNT_PROPERTY`**

Constant value: **`profileCount`**

MODELMAP_RELATIONCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_RELATIONCOUNT_PROPERTY`**

Constant value: **`relationCount`**

MODELMAP_REQUIREMENTCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_REQUIREMENTCOUNT_PROPERTY`**

Constant value: **`requirementCount`**

MODELMAP_SEQUENCEDIAGRAMCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_SEQUENCEDIAGRAMCOUNT_PROPERTY`**

Constant value: **`sequenceDiagramCount`**

MODELMAP_SOURCEELEMENTSCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_SOURCEELEMENTSCOUNT_PROPERTY`**

Constant value: **`sourceElementsCount`**

MODELMAP_STATECHARTCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_STATECHARTCOUNT_PROPERTY`**

Constant value: **`statechartCount`**

MODELMAP_STATECHARTDIAGRAMCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_STATECHARTDIAGRAMCOUNT_PROPERTY`**

(continued from last page)

Constant value: **`statechartDiagramCount`**

MODELMAP_STEREOTYPECOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_STEREOTYPECOUNT_PROPERTY`**

Constant value: **`stereotypeCount`**

MODELMAP_STRUCTUREDIAGRAMCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_STRUCTUREDIAGRAMCOUNT_PROPERTY`**

Constant value: **`structureDiagramCount`**

MODELMAP_TABLEVIEWCOUNT_PROPERTY

`public static final java.lang.String` **`MODELMAP_TABLEVIEWCOUNT_PROPERTY`**

Constant value: **`tableViewCount`**

Constructors

ModelMapController

`public` **`ModelMapController`**`()`

controller
Class PyramidController

```
java.lang.Object
   |-
      |-controller.PyramidController
```

All Implemented Interfaces:
java.beans.PropertyChangeListener

---

public class **PyramidController**
extends AbstractController

MVC Controller for the Overview Pyramid functionality of the Model Meter. Acts as a mediator between the registered models and the views.

Fields

PYRAMID_AHHCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_AHHCOUNT_PROPERTY`**

Constant value: **`AhhCount`**

PYRAMID_AHHINTERPRETATION_PROPERTY

`public static final java.lang.String` **`PYRAMID_AHHINTERPRETATION_PROPERTY`**

Constant value: **`AhhInterpretation`**

PYRAMID_ANDCCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_ANDCCOUNT_PROPERTY`**

Constant value: **`AndcCount`**

PYRAMID_ANDCINTERPRETATION_PROPERTY

`public static final java.lang.String` **`PYRAMID_ANDCINTERPRETATION_PROPERTY`**

Constant value: **`AndcInterpretation`**

PYRAMID_CYCLOCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_CYCLOCOUNT_PROPERTY`**

Constant value: **`CycloCount`**

(continued from last page)

PYRAMID_CYCLOLOCINTERPRETATION_PROPERTY

`public static final java.lang.String` **`PYRAMID_CYCLOLOCINTERPRETATION_PROPERTY`**

Constant value: **`CycloLocInterpretation`**

PYRAMID_CYCLOLOCRATIO_PROPERTY

`public static final java.lang.String` **`PYRAMID_CYCLOLOCRATIO_PROPERTY`**

Constant value: **`CycloLocRatio`**

PYRAMID_LOCCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_LOCCOUNT_PROPERTY`**

Constant value: **`LocCount`**

PYRAMID_LOCNOSEINTERPRETATION_PROPERTY

`public static final java.lang.String` **`PYRAMID_LOCNOSEINTERPRETATION_PROPERTY`**

Constant value: **`LocNoseInterpretation`**

PYRAMID_LOCNOSERATIO_PROPERTY

`public static final java.lang.String` **`PYRAMID_LOCNOSERATIO_PROPERTY`**

Constant value: **`LocNoseRatio`**

PYRAMID_NOCCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_NOCCOUNT_PROPERTY`**

Constant value: **`NocCount`**

PYRAMID_NOCNOPINTERPRETATION_PROPERTY

`public static final java.lang.String` **`PYRAMID_NOCNOPINTERPRETATION_PROPERTY`**

Constant value: **`NocNopInterpretation`**

PYRAMID_NOCNOPRATIO_PROPERTY

`public static final java.lang.String` **`PYRAMID_NOCNOPRATIO_PROPERTY`**

Constant value: **`NocNopRatio`**

PYRAMID_NOEGCOUNT_PROPERTY

`public static final java.lang.String` **`PYRAMID_NOEGCOUNT_PROPERTY`**

(continued from last page)

Constant value: **CallsCount**

PYRAMID_NOEGNOSEINTERPRETATION_PROPERTY

`public static final java.lang.String` **PYRAMID_NOEGNOSEINTERPRETATION_PROPERTY**

Constant value: **CallsNoseInterpretation**

PYRAMID_NOEGNOSERATIO_PROPERTY

`public static final java.lang.String` **PYRAMID_NOEGNOSERATIO_PROPERTY**

Constant value: **CallsNoseRatio**

PYRAMID_NOPCOUNT_PROPERTY

`public static final java.lang.String` **PYRAMID_NOPCOUNT_PROPERTY**

Constant value: **NopCount**

PYRAMID_NORCOUNT_PROPERTY

`public static final java.lang.String` **PYRAMID_NORCOUNT_PROPERTY**

Constant value: **NorCount**

PYRAMID_NORNOEGINTERPRETATION_PROPERTY

`public static final java.lang.String` **PYRAMID_NORNOEGINTERPRETATION_PROPERTY**

Constant value: **NorCallsInterpretation**

PYRAMID_NORNOEGRATIO_PROPERTY

`public static final java.lang.String` **PYRAMID_NORNOEGRATIO_PROPERTY**

Constant value: **NorCallsRatio**

PYRAMID_NOSECOUNT_PROPERTY

`public static final java.lang.String` **PYRAMID_NOSECOUNT_PROPERTY**

Constant value: **NoseCount**

PYRAMID_NOSENOCINTERPRETATION_PROPERTY

`public static final java.lang.String` **PYRAMID_NOSENOCINTERPRETATION_PROPERTY**

Constant value: **NoseNocInterpretation**

PYRAMID_NOSENOCRATIO_PROPERTY

```
public static final java.lang.String PYRAMID_NOSENOCRATIO_PROPERTY
```

Constant value: **NoseNocRatio**

PYRAMID_PROGRESSSTEPCOUNT_PROPERTY

```
public static final java.lang.String PYRAMID_PROGRESSSTEPCOUNT_PROPERTY
```

Constant value: **PorgressStepCount**

PYRAMID_PROGRESSSTEPINDEX_PROPERTY

```
public static final java.lang.String PYRAMID_PROGRESSSTEPINDEX_PROPERTY
```

Constant value: **PorgressStepIndex**

Constructors

PyramidController

```
public PyramidController()
```

Package
helper

helper
Class BrowserCaller

```
java.lang.Object
   |
   +-helper.BrowserCaller
```

public class **BrowserCaller**
extends Object

Provides system independent functionality for calling documents with enviroment`s browser.

Constructors

BrowserCaller

```
public BrowserCaller()
```

Methods

openLocalFileRelativeToUserDir

```
public void openLocalFileRelativeToUserDir(String aFileName)
```

Open a file relative to the apllication directory in the enviroment`s browser.

Parameters:
`aFileName` - the file to be opened in the browser

helper
Class StringHelper

```
java.lang.Object
   |
   +-helper.StringHelper
```

public class **StringHelper**
extends Object

This class provides helper methods for working with strings.

Constructors

StringHelper

```
public StringHelper()
```

Methods

isNullOrEmpty

```
public boolean isNullOrEmpty(String str)
```

This method returns true if the given string is empty or null.

Parameters:
`str` - the string to be analysed

Returns:
true if the given string is empty or null; false otherwise

containsListElement

```
public int containsListElement(List elements,
         String subject)
```

Counts the appearance of list elements in the given text. Every time an element appears more than one time the count is incremented.

Parameters:
`elements` - List of String elements
`subject` - String to be analysed

Returns:
the count of list elements that are at least on time conatining in the given subject string

cutDecimalPlace

```
public String cutDecimalPlace(String decimal)
```

This method will cut the decimal places if they are "0" from a given String in decimal format.

(continued from last page)

Parameters:

`decimal` - decimal number as a string

Returns:

string with cut decimal places

Package
main

main
Class ModelMeterFactory

```
java.lang.Object
   |
   +-main.ModelMeterFactory
```

public class **ModelMeterFactory**
extends Object

Holds all instances of the Modal Meter as compositions. Creates the instances of its composited childs. Initializes the MVC logic and the application views. Provides reference getters for the instance of the Model Meter application. Provides a desctructor which disposes the dialogs for the garbage collection context.

Constructors

ModelMeterFactory

```
public ModelMeterFactory(IRPApplication app)
```

Methods

finalize

```
public void finalize()
```

Desctructor which diposes the dialogs and the main frame view.

getOverviewPyramid

```
public OverviewPyramid getOverviewPyramid(String language)
```

Gets an overview pyramid reference by implementation language.

Parameters:
`language` - the language of the reference model the returned pyramid should be

Returns:
the overview pyramid for the given reference language

getProcessingDialog

```
public javax.swing.JDialog getProcessingDialog()
```

Returns:
the processing dialog view

getMainFrame

```
public javax.swing.JFrame getMainFrame()
```

(continued from last page)

Returns:
the main view

getPyramidController

```
public PyramidController getPyramidController()
```

Returns:
the pyramid controller

getPyramidPanel

```
public PyramidPanel getPyramidPanel()
```

Returns:
the pyramid panel view

getProcessingStatePanel

```
public ProcessingStatePanel getProcessingStatePanel()
```

Returns:
the processing state panel view

getModelMap

```
public ModelMap getModelMap()
```

Returns:
the model map model

getModelMeterFlowControl

```
public ModelMeterFlowControl getModelMeterFlowControl()
```

Returns:
the flow control of the model meter

getLanguageChooserDialog

```
public javax.swing.JDialog getLanguageChooserDialog()
```

Returns:
the language chooser dialog view

getLanguageChooserPanel

```
public LanguageChooserPanel getLanguageChooserPanel()
```

Returns:
the language chooser panel view

getConfigurationModel

```
public ConfigurationModel getConfigurationModel()
```

Returns:
the configuration model

getBasicValuesPanel

```
public BasicValuesPanel getBasicValuesPanel()
```

Returns:
the basic values panel view

getDisharmonyDetector

```
public DisharmonyDetectionModel getDisharmonyDetector()
```

Returns:
the disharmony detector

getAboutDialog

```
public AboutDialog getAboutDialog()
```

Returns:
the about dialog view

main
Class ModelMeterFlowControl

```
java.lang.Object
   |
   +-main.ModelMeterFlowControl
```

public class **ModelMeterFlowControl**
extends Object

Controls the application`s flow regarding its gui behaviour.

Constructors

ModelMeterFlowControl

```
public ModelMeterFlowControl(ModelMeterFactory factory,
                             IRPApplication app)
```

Methods

processSelectedLanguageOnElement

```
public void processSelectedLanguageOnElement()
```

Performs a measurement depending on the choosen reference language.

processSelectedElement

```
public void processSelectedElement()
```

Performs a measurement on the actual selected element.

processInvocationCall

```
public void processInvocationCall(IRPModelElement modelElement)
```

Processes a rhaposdy invocation call. Performs a measurement depending on the configured reference language.

Parameters:
`modelElement` - the model element to be measured recursively

checkReferenceLanguage

```
public void checkReferenceLanguage()
```

Opens a chooser dialog for the reference language if it is undefined in the configuration.

setAboutDialogVisible

```
public void setAboutDialogVisible(boolean aIsVisible)
```

(continued from last page)

Sets the about dialog in-/visible.

Parameters:

`aIsVisible` - true show the dialog; false hide the dialog

Package
metrics

metrics
Class ComplexityCounter

```
java.lang.Object
   |
   +-metrics.ComplexityCounter
```

public class **ComplexityCounter**
extends Object

Provides methods for counting Mc Cabe's cyclomatic complexity of flowcharts (Statecharts/Activitydiagrams) of a given Rhapsody Model.

Constructors

ComplexityCounter

```
public ComplexityCounter()
```

Methods

getComplexity

```
public int getComplexity(ModelMap aModelMap)
```

Counts the overall cyclomatic complexity regarding the flowcharts of a given ModelMap-Object.

Parameters:
`aModelMap` - the mode map as the measurement base

Returns:
the cyclomatic complexity of the flowcharts

getComplexityOfStateChart

```
public int getComplexityOfStateChart(IRPStatechart aStateChart)
```

Counts the given statecharts cyclomatic complexity.

Parameters:
`aStateChart` - the statechart/activity diagram to be measured

Returns:
the cyclomatic complexity of the statechart/activity diagram

metrics
Class CouplingCounter

```
java.lang.Object
   |
   +-metrics.CouplingCounter
```

public class **CouplingCounter**
extends Object

Provides functionality to count coupling metrics for a given Rhapsody model.

Constructors

CouplingCounter

```
public CouplingCounter()
```

Methods

getNumberOfEventGenerations

```
public int getNumberOfEventGenerations(ModelMap map)
```

Returns the number of event generations for the given model map. If an event is generated at least for one time, the count will be incremented by one.

Parameters:
`map` - the model to be analysed

Returns:
the count of event generations

metrics
Class InheritanceCounter

```
java.lang.Object
   |
   +-metrics.InheritanceCounter
```

public class **InheritanceCounter**
extends Object

Provides methods for counting inheritance aspects for a given Rhapsody Model.

Constructors

InheritanceCounter

```
public InheritanceCounter()
```

Methods

getAverageNumberOfDerivedClasses

```
public double getAverageNumberOfDerivedClasses(ModelMap aModelMap)
```

Returns the average number of derived classes for the given ModelMap-Object.

Parameters:
`aModelMap` - the model map to be analysed

Returns:
the average number of derived classes for the given model map

getAverageHierarchyHeight

```
public double getAverageHierarchyHeight(ModelMap aModelMap)
```

Returns the average inheritance hierarchy height of the given ModelMap-Object.

Parameters:
`aModelMap` - the model map to be analysed

Returns:
the average hierarchy height

metrics
Class LocCounter

```
java.lang.Object
   |
   +-metrics.LocCounter
```

public class **LocCounter**
extends Object

Provides methods for counting the physical lines of code for C/C++/Java. SLOC = physical, non-comment lines.

Constructors

LocCounter

```
public LocCounter()
```

Methods

getLocCount

```
public int getLocCount(ModelMap aModelMap)
```

Counts the overall slocs for a given ModelMap-Reference.

Parameters:
`aModelMap` - the model map to be analysed

Returns:
the loc count

metrics
Class OMDCounter

```
java.lang.Object
   |
   +-metrics.OMDCounter
```

public class **OMDCounter**
extends Object

This class provides methods for the calculating for calculating a specific object model diagram`s complexity, based on the idea of the rule of seven.

Constructors

OMDCounter

```
public OMDCounter()
```

Methods

getNumberOfOmdComplexity

```
public int getNumberOfOmdComplexity(IRPObjectModelDiagram omd)
```

Calculates the specific complexity for an object model diagram, based on the idea of the rule of seven.

Parameters:
`omd` - the object model diagram to be analysed

Returns:
the complexity of the object model diagram

Package
model

model
Class AbstractModel

```
java.lang.Object
   |
   +-model.AbstractModel
```

Direct Known Subclasses:
ConfigurationModel, DisharmonyDetectionModel, ModelMap, OverviewPyramid

---

public abstract class **AbstractModel**
extends Object

This class provides base level functionality for all models, including a support for a property change mechanism (using the PropertyChangeSupport class), as well as a convenience method, that other objects can use to reset model state.

Fields

propertyChangeSupport

```
protected java.beans.PropertyChangeSupport propertyChangeSupport
```

Constructors

AbstractModel

```
public AbstractModel()
```

Default constructor. Instantiates the PropertyChangeSupport class.

Methods

addPropertyChangeListener

```
public void addPropertyChangeListener(java.beans.PropertyChangeListener listener)
```

Adds a property change listener to the observer list.

Parameters:
`listener` - the property change listener

---

removePropertyChangeListener

```
public void removePropertyChangeListener(java.beans.PropertyChangeListener listener)
```

Removes a property change listener from the observer list.

Parameters:
`listener` - the property change listener

(continued from last page)

firePropertyChange

```
protected void firePropertyChange(String propertyName,
          Object oldValue,
          Object newValue)
```

Fires an event to all registered listeners, informing them that a property in this model has changed.

Parameters:

`propertyName` - the name of the property
`oldValue` - the previous value of the property before the change
`newValue` - the new property value after the change

model
Class ConfigurationModel

```
java.lang.Object
   |
   +-
      |
      +-model.ConfigurationModel
```

public class **ConfigurationModel**
extends AbstractModel

Provides getters/setters for the application`s basic configuration values.

Constructors

ConfigurationModel

```
public ConfigurationModel()
```

Methods

getReferenceLanguage

```
public String getReferenceLanguage()
```

Returns:
the reference language

setReferenceLanguage

```
public void setReferenceLanguage(String aReferenceLanguage)
```

Sets the reference language for the metric model.

Parameters:
`aReferenceLanguage` - the reference language to be setted

initDefaults

```
public void initDefaults()
```

Initializes the model with default values.

getBuildLabel

```
public String getBuildLabel()
```

Returns:

(continued from last page)

the build label

model
Class DisharmonyDetectionModel

```
java.lang.Object
   |
   +-
      |
      +-model.DisharmonyDetectionModel
```

public class **DisharmonyDetectionModel**
extends AbstractModel

This class represents the model of disharmony detections for a given model map.

Constructors

DisharmonyDetectionModel

```
public DisharmonyDetectionModel()
```

Methods

doDetection

```
public void doDetection(ModelMap modelMap)
```

Executes the disharmony detection.

Parameters:
`modelMap` - the model map to be analysed

setMcCabeThreshold

```
public void setMcCabeThreshold(Integer mcCabeThreshold)
```

Parameters:
`mcCabeThreshold` - the McCabe threshold to be setted

setOmdThreshold

```
public void setOmdThreshold(Integer omdThreshold)
```

Parameters:
`omdThreshold` - the object model threshold to be setted

model
Class ModelMap

```
java.lang.Object
   |
   +-
      |
      +-model.ModelMap
```

public class **ModelMap**
extends AbstractModel

Represents the metric regarding properties from a captured Rhapsody Model. Provides getter-methods for those properties.

Constructors

ModelMap

```
public ModelMap()
```

Methods

captureModel

```
public void captureModel(IRPModelElement aModelElement,
         IRPApplication app)
```

Captures the model under the given Rhapsody Model Element. Counts elements. Adds elements to lists necessary for getting metric counts. For iterating the elements only the IRPCollection is used. The conversion toList() takes as long as it takes catching the elements by index but no progress could be reported. The IRPCollection index runs from 1 to getCount().

Parameters:

`aModelElement` - - the model map to be analysed
`app` - the reference to the rhapsody application instance

getRootModelElementClass

```
public String getRootModelElementClass()
```

Returns:

the root model element class

getClassAndObjectCount

```
public int getClassAndObjectCount()
```

Components and modules are also counted as classes via the Rhaspody model. In this case only code generating elements are counted. Constraint: There have been cases with old projects where components have been counted as classes.

Returns:

the class count

getPackageCount

```
public int getPackageCount()
```

Returns:
the package count

getStatechartList

```
public List getStatechartList()
```

Returns:
the statechart list

getRootModelElementName

```
public String getRootModelElementName()
```

Returns:
the root model element name

getClassList

```
public List getClassList()
```

Returns:
the class list

getSourceElementsCount

```
public int getSourceElementsCount()
```

Returns:
the source element count

getOmdList

```
public List getOmdList()
```

Returns:
the object model diagram list

getSourceElements

```
public List getSourceElements()
```

(continued from last page)

Returns:
the source elements

getEvents

```
public List getEvents()
```

Returns:
the event list

getRelationCount

```
public int getRelationCount()
```

Returns:
the relation count

getProjectName

```
public String getProjectName()
```

Returns:
the project name

getApplicationLanguage

```
public String getApplicationLanguage()
```

Returns:
the application language

getApplicationBuildNo

```
public String getApplicationBuildNo()
```

Returns:
the application build number

getCurrentDirectory

```
public String getCurrentDirectory()
```

Returns:
the current project directory

getProjectLanguage

```
public String getProjectLanguage()
```

Returns:
the project language

model
Class OverviewPyramid

```
java.lang.Object
   |
   +-
      |
      +-model.OverviewPyramid
```

public class **OverviewPyramid**
extends AbstractModel

Represents the overview pyramid model. Provides methods for executing the measurement of the metric counts for a given ModelMap-Object. Calculates the pyramid Ratios and Interpretations.

Constructors

OverviewPyramid

```
public OverviewPyramid()
```

Methods

doMeasure

```
public void doMeasure(ModelMap aModelMap)
```

Measures the pyramid metrics, calculates the ratios between the determined values and retrieves the ratio interpretations for a given ModelMap-Object.

Parameters:
`aModelMap` - the model map to be measured

getLanguage

```
public String getLanguage()
```

Returns:
the pyramid's reference language

model
Class OverviewPyramids

```
java.lang.Object
   |
   +-model.OverviewPyramids
```

public class **OverviewPyramids**
extends Object

Holds the instances for the configured overview pyramid by their reference language. Provides methods for getting a pyramid by its language.

Constructors

OverviewPyramids

```
public OverviewPyramids()
```

Methods

getOverviewPyramid

```
public OverviewPyramid getOverviewPyramid(String language)
```

Returns an overview pyramid instance by the given refernece language.

Parameters:
`language` - the reference language model

Returns:
the overview pyramid for the reference model language

getLanguages

```
public List getLanguages()
```

Returns configured pyramid reference languages.

Returns:
the list of available reference language models

model
Class OverviewPyramidThresholdSet

```
java.lang.Object
   |
   +-model.OverviewPyramidThresholdSet
```

public class **OverviewPyramidThresholdSet**
extends Object

Represents an overview pyramid thresholdset for metric correlation.

Constructors

OverviewPyramidThresholdSet

```
public OverviewPyramidThresholdSet()
```

Methods

getRatioInterpretation

```
public String getRatioInterpretation(double ratio)
```

Returns the interpretation (e.g. low, average, high) for the given ratio.

Parameters:
`ratio` - the ratio to be interpreted

Returns:
the interpretation out of low, high, average

getMetricCorrelation

```
public String getMetricCorrelation()
```

Returns:
the metric correlation

Package
view

view
Class AboutDialog

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.AboutDialog
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.accessibility.Accessible, javax.swing.TransferHandler.HasGetTransferHandler, javax.swing.RootPaneContainer, javax.accessibility.Accessible, javax.swing.WindowConstants

---

public class **AboutDialog**
extends javax.swing.JDialog

This implements a dialog container for the about panel.

Constructors

AboutDialog

```
public AboutDialog(java.awt.Frame owner)
```

Dialogs constructor.

Parameters:
`owner` - the dialogs owner frame

view
Class AboutPanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.AboutPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **AboutPanel**
extends AbstractViewPanel

This implements a swing panel for showing basic "about" information of the application itself.

Constructors

AboutPanel

```
public AboutPanel(javax.swing.JDialog parentDialog)
```

Panels constructor.

Parameters:
`parentDialog` - the panels owner dialog

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class AbstractViewPanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-view.AbstractViewPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

Direct Known Subclasses:
AboutPanel, BasicValuesPanel, LanguageChooserPanel, McCabePanel, ProcessingStatePanel, PyramidPanel, RuleOfSevenPanel

---

public abstract class **AbstractViewPanel**
extends javax.swing.JPanel

This class provides the base level abstraction for views in this example. All views that extend this class also extend JPanel (which is useful for performing GUI manipulations on the view in NetBeans Matisse), as well as providing the modelPropertyChange() method that offers controllers the opportunity to propogate model property changes.

Constructors

AbstractViewPanel

```
public AbstractViewPanel()
```

Methods

modelPropertyChange

```
public abstract void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

Called by the controller when it needs to pass along a property change from a model.

Parameters:
`evt` - the property change event from the model

view
Class BasicValuesPanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.BasicValuesPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **BasicValuesPanel**
extends AbstractViewPanel

This implements a swing panel for showing a table with the basic values of the selected model element.

Constructors

BasicValuesPanel

```
public BasicValuesPanel()
```

Panels constructor.

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class LanguageChooserDialog

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.LanguageChooserDialog
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.accessibility.Accessible, javax.swing.TransferHandler.HasGetTransferHandler, javax.swing.RootPaneContainer, javax.accessibility.Accessible, javax.swing.WindowConstants

---

public class **LanguageChooserDialog**
extends javax.swing.JDialog

This implements a dialog container for the language chooser panel.

Constructors

LanguageChooserDialog

```
public LanguageChooserDialog(java.awt.Frame owner)
```

Dialogs constructor.

Parameters:
`owner` - the dialogs owner frame

view
Class LanguageChooserPanel

```
java.lang.Object
   |-
      |-
         |-
            |-
               |-
                  |-view.LanguageChooserPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **LanguageChooserPanel**
extends AbstractViewPanel

This implements a swing panel for choosing the metrics reference language model.

Constructors

LanguageChooserPanel

```
public LanguageChooserPanel(ConfigurationController controller,
                            javax.swing.JDialog parentDialog,
                            List referenceLanguages)
```

Panles constructor.

Parameters:
`controller` - the configurations model controller
`parentDialog` - the panels owner dialog
`referenceLanguages` - the actual reference language

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class MainView

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.MainView
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.accessibility.Accessible, java.awt.MenuContainer, javax.swing.TransferHandler.HasGetTransferHandler, javax.swing.RootPaneContainer, javax.accessibility.Accessible, javax.swing.WindowConstants

---

public class **MainView**
extends javax.swing.JFrame

This implements a frame panel as the application's main container with a menue bar and a treeview for selecting the content panels.

Constructors

MainView

```
public MainView(ModelMeterFlowControl flowControl,
                PyramidPanel pyramidPanel,
                BasicValuesPanel basicValuesPanel,
                McCabePanel mcCabePanel,
                RuleOfSevenPanel ruleOfSevenPanel)
```

Frames constructor.

Parameters:
`flowControl` - the flow control of model meter
`pyramidPanel` - the pyramid panel
`basicValuesPanel` - the basic value panel
`mcCabePanel` - the Mc Cabe panel
`ruleOfSevenPanel` - the Rule of Seven panel

view
Class McCabePanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.McCabePanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **McCabePanel**
extends AbstractViewPanel

This implements a swing panel for showing a list of satecharts/activity diagrams of a given Mc Cabe complexity. The complexity can be adjusted via a slider.

Constructors

McCabePanel

```
public McCabePanel(DisharmonyController disharmonyController)
```

Panels constructor.

Parameters:
`disharmonyController` - the controller of the disharmony model

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class ProcessingStateDialog

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.ProcessingStateDialog
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.accessibility.Accessible, javax.swing.TransferHandler.HasGetTransferHandler, javax.swing.RootPaneContainer, javax.accessibility.Accessible, javax.swing.WindowConstants

---

public class **ProcessingStateDialog**
extends javax.swing.JDialog

This implements a swing dialog as container for the processing state panel.

Constructors

ProcessingStateDialog

```
public ProcessingStateDialog(java.awt.Frame owner)
```

Dialogs constructor.

Parameters:
`owner` - the dialogs owner frame

view
Class ProcessingStatePanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.ProcessingStatePanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **ProcessingStatePanel**
extends AbstractViewPanel

This implements a swing panel for showing the processing state while the measuring and analysis are taking place.

Constructors

ProcessingStatePanel

```
public ProcessingStatePanel(javax.swing.JDialog parentDialog)
```

Panels constructor.

Parameters:
`parentDialog` - the panels owner dialog

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class PyramidPanel

```
java.lang.Object
   ┣-
      ┣-
         ┣-
            ┣-
               ┣-
                  ┣-
                     ┣-view.PyramidPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **PyramidPanel**
extends AbstractViewPanel

This implements a swing panel for showing the metrics overview pyramid.

Constructors

PyramidPanel

```
public PyramidPanel()
```

Panels constructor.

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

view
Class RuleOfSevenPanel

```
java.lang.Object
   ┗-
      ┗-
         ┗-
            ┗-
               ┗-
                  ┗-view.RuleOfSevenPanel
```

All Implemented Interfaces:
Serializable, java.awt.MenuContainer, java.awt.image.ImageObserver, javax.swing.TransferHandler.HasGetTransferHandler, Serializable, javax.accessibility.Accessible

---

public class **RuleOfSevenPanel**
extends AbstractViewPanel

This implements a swing panel for showing a list of object model diagrams with a given complexity. The complexity can be adjusted by a slider.

Constructors

RuleOfSevenPanel

```
public RuleOfSevenPanel(DisharmonyController disharmonyController)
```

Panels constructor.

Methods

modelPropertyChange

```
public void modelPropertyChange(java.beans.PropertyChangeEvent evt)
```

This method provides the interface for property changes of models via the applications MVC logic.

Parameters:
`evt` - the property change event from the model

Anhang F

Model Meter CD Contents

Contents of the attached Model Meter CD

```
CD
|_Annex
| |
| |_Abstract.pdf           - Masterthesis abstract
| |
| |_BuildInstructions.pdf  - How to build and run the Model Meter source
| |
| |_MeasurementResults.pdf - Measurement results of the 32 C++ projects
| |
| |_ModelMeterApiDoc.pdf   - API Documentation of Model Meter
| |
| |_ModelMeterManual.pdf   - Details of regarding installation and usage
| |
| |_ValidationSheet.pdf    - Interview sheet for reference model validation
|
|_Environment              - The environment the work was built in
|_
| |_Doc                    - OpenOffice.org 3.1.1
|_
| |_Eclipse                - Galileo and plugins
|_
| |_Java                   - JRE, JDK, Libraries and doclets
|_
| |_LaTeX                  - MiKTeX, JabRef, TeXnicCenter, ...
|
|_Masterthesis             - PDF Document and its latex sources
|
|_ModelMeter               - The product itself
| |
| |_cfg                    - Configuration files of the plugin
| | |
| | |_configuration.xml    - General settings
| | |
| | |_logging.properties   - Logging behaviour
| | |
| | |_overviewPyramids.xml - Overview pyramid threshold sets
| |
| |_help                   - HTML Version of the manual
| |
| |_log                    - Log files of the plugin
| |
| |_ModelMeter.jar         - The compiled plugin
| |
| |_ModelMeter.hep         - Helper file for rhapsody integration
|
|_Source
| |
| |_Documentation          - The open office source for the annex pdfs
| |
| |_JavaAPIDoc             - The API documentation
| |
| |_ModelMeter             - The java source/eclipse project/libs/...
| |
| |_RhapPluginTest         - The rhapsody project for the unit tests
```

```
Note: The environment is exluding MS Visio 2007 Pro and IBM Rhapsody 7.5.0.
      Please get valid licences for their usage.
```

Anhang G

Glossar

AUTOSAR Ein internationaler Verbund von Automobilherstellern, mit dem Ziel, einen offenen Architekturstandard für Software einzurichten.

Bad Smell Auch *Code Smell* oder *Anti Pattern* genannt ist ein Indikator um in Quellcode Probleme ausfindig zu machen [Fow99].

DoDAF Referenzarchitekturmodell zur Erstellung von Anwendungen des U.S. Department of Defense.

MODAF Referenzarchitekturmodell zur Erstellung von Anwendungen des UK Ministry of Defence.

GUI Mockup Konzeptskizze einer grafischen Benutzeroberfläche. Wird meist in frühen Designphasen zur Kommunikation von Vorstellungen genutzt.

Silver Bullet SW-Technik, die produktive Lösungen für die folgenden vier grundlegenden Probleme der Software-Entwicklung bietet: Hohe Software-Komplexität, Konformität, Änderungsfähigkeit und Unsichtbarkeit [Boe08]. Diese wurden in [Bro86] als nicht einheitlich lösbar postuliert.

Stakeholder In der Software-Technik eine Person, welche Anforderungen an ein zu erstellendes Produkt besitzt.

SysML standardisierte Sprache und grafische Notation zur System-Modellierung. Basiert auf der UML.

UML Von der Object Management Group standardisierte Sprache und grafische Notation für die Modellierung von Software.

Literaturverzeichnis

[Abr95] *Toward the Design Quality Evaluation of Object-Oriented Software Systems.* 1995

[AKP02] Assmann, Danilo ; Kalmar, Ralf ; Punter, Dr. T.: Messen und Bewerten von Webapplikationen mit der Goal/Question/Metric Methode / Fraunhofer Institut für experiementelles Software Engineering. 2002. – Forschungsbericht

[Alb79] *Measuring Application Development Productivity.* 1979

[Apa09] Apache: *log4j.* 2009. – `logging.apache.org/log4j`

[AV04] Andersson, Magnus ; Vestergren, Patrik: *Object-Oriented Design Quality Metrics*, Uppsala University Sweden, Diplomarbeit, 2004

[Bal98] Balzert, Helmut: *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 1998

[BBK+78] Boehm, Barry ; Brown, J.R. ; Kaspar, H. ; Lipow, M. ; McLeod, G. ; Merritt, M.: *Characteristics of Software Quality.* North Holland, 1978

[BD02] Bansiya, Jagdish ; Davis, Carl G.: A Hierarchical Model for Object-Oriented Design Quality Assessment. In: *IEEE Transactions on Software Engineering* 28 (2002)

[Boe08] Boehm, Barry: Das Software-Engineering im 20. und 21. Jahrhundert. In: *OBJEKTSpektrum* 1 (2008), November / Dezember, Nr. 6, S. 16 – 25

[Bro86] Brooks, Fredericks P.: No Silver Bullet - essence and accident in software Engineering. In: *Proceedings of the IFIP Tenth World Computing Conference* (1986), S. 1069–1076

[BSL99] Basili, V. ; Shull, F. ; Lanubille, F.: Building Knowledge through families of experiments. In: *IEEE Transactions on Software Engineering* 25 (1999)

[Bur09] Burn, Oliver: *Checkstyle.* 2009. – `checkstyle.sourceforge.net`

[BW84] Basili, V. ; Weiss, D.: A Methodology for Collecting Valid Software Engineering Data. In: *IEEE Transactions on Software Engineering* 10 (1984)

[DDN08] Demeyer, Serge ; Ducasse, Stéphane ; Nierstrasz, Oscar: *Object-Oriented Reengineering Patterns*. First Open-Source Edition. Square Bracket Associates, 2008

[DeM09] DeMarco, Tom: Software engineering: An Idea Whose Time Has Come and Gone? In: *IEEE Sofware* (2009), S. 94–95

[Eck07] Eckstein, Robert: Java SE Application Design With MVC. In: *Sun Developer Network (SDN)* (2007). – `java.sun.com/developer/technicalArticles/javase/mvc`

[EWEBBF04] El-Wakil, Mohamed ; El-Bastawisi, Ali ; Boshra, Mokhtar ; Fahmy, Ali: Object-Oriented Design Quality Models - A Survey and Comparison / Faculty of Computers and Information,Cairo University, Egypt. 2004. – Forschungsbericht

[Fow99] Fowler, Martin: *Refactoring - Improving the Design of existing Code*. Addison-Wesley, 1999

[FP96] Fenton, Norman E. ; Pfleeger, Shari L.: *Software Metrics - A Rigorous & Practical Approach*. International Thomson Computer Press, 1996

[Gal09] Gallagher, Niall: *Simple XML serialization*. 2009. – `simple.sourceforge.net`

[Gen02] *Defining and Validating Metrics for UML Statechart Diagrams*. 2002

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elemets of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (Professional Computing Series)

[Gil77] Gilb, Tom: *Software Metrics*. Winthrop Publishers, Inc., 1977

[GMP08] Görtz, Thomas ; Macke, Stefan ; Peisker, Patrick: Software-Architekturmuster / Georg-Simon-Ohm-Hochschule Nürnberg. 2008. – Forschungsbericht

[Goc08] Gockel, Tilo: *Form der wissenschaftlichen Ausarbeitung*. Berlin Heidelberg : Springer-Verlag, 2008. – Begleitende Materialien unter `www.formbuch.de`

[Hal77] Halstead, Maurice H.: *Elements of software science*. Elsevier, 1977

[Har87] Harel, David: Statecharts: A visual formalism for complex systems. In: *Science of Computer Programming* 8 (1987), S. 231–274

[Hau09] Haug, Thomas: *Software Metriken*. 2009. – `www.mathema.de`

[Hop08] Hopf, Prof. Dr. Hans-Georg: *Softwarequalität - Ausgewählte Themen*. 1. Nürnberg : Georg-Simon-Ohm-Fachhochschule, 2008

[HS96] Henderson-Sellers, Brian: *Object-Oriented Metrics*. Prentice Hall PTR, 1996

[JL07] Janssen, Jürgen ; Laatz:, Wilfried: *Statistische Datenanalyse mit SPSS für Windows*. Springer, 2007

[JUn09] JUnit.org: *JUnit*. 2009. – `www.junit.org`

[Kan03] Kan, Stephen H.: *Metrics and Models in Software Quality Engineering*. Second Edition. Addison-Wesley, 2003

[KB02] Kamiske, Gerd F. ; Brauer, Jörg-Peter: *ABC des Qualitäts-Managements*. Carl Hanser Verlag, 2002

[Kee06] *Automated Design Improvement by Example*. 2006

[Lan09] Lanza, Michele: *Episode 130: Code Visualization with Michele Lanza*. Software Engineering Radio, 2009

[LC08] Lokietsch, Martin ; Czada, Sonja: *The Rhapsody Story*. IBM Rational, 2008

[LH89] Lieberherr, Karl J. ; Holland, I.: Assuring good style for object-oriented programs. In: *IEEE Sofware* (1989), S. 38–48

[LK94] Lorenz, Mark ; Kidd, Jeff: *Object-Oriented Software Metrics*. Prentice Hall PTR, 1994

[LM06] Lanza, Michele ; Marinescu, Radu: *Object-Oriented Metrics in Practice*. Berlin Heidelberg : Springer-Verlag, 2006

[LW07] Lanza, Michele ; Wettel, Richard: CodeCity 3D Visualization of Large-Scale Software. (2007)

[Mar08] Martin, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall International, 2008

[McC76] McCabe, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* 2 (1976), S. 1976

[Mic97] Microsystems, Sun: Java Code Conventions. (1997)

[Mil56] Miller, George A.: The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. In: *Psychological Review* 63 (1956), S. 81–97

[Mor89] Morris, Kenneth L.: *Metrics for Object-Oriented Software Development Environments*, MIT, Diplomarbeit, 1989

[Pin97] Pinker, Steven: *How the Mind works*. Penguin Books, 1997

[PT07] Pietrek, Georg ; Trompeter, Jens: *Modellgetriebene Softwareemtwicklung*. entwickler.press, 2007

[Rie96] Riel, Arthur J.: *Object-Oriented Design Heuristics*. Addison-Wesley, 1996

[Roc88] Rocacher, Daniel: *Metrics Definition for Smalltalk*. MUSE WP9A, 1988

[RQZ07] Rupp, Chris ; Queins, Stefan ; Zengler, Barbara: *UML 2 Glasklar*. München : Carl Hanser Verlag, 2007

[SB99] Solingen, Rini van ; Berghout, Egon: *The Goal/Question/Metric Method: A practical guide for quality improvement of software development*. The McGraw-Hill Companies, 1999

[SRK94] Shyam R, Chidamber ; Kemerer, Chris: A Metric Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), Juni, Nr. 6, S. 476 – 493

[SSM06] Simon, Frank ; Seng, Olaf ; Mohaupt, Thomas: *Code-Qualitymanagement*. dpunkt.verlag, 2006

[SVH07] Stahl, Thomas ; Völter, Markus ; Haase, Arno: *Modellgetriebene Software-Entwicklung*. dpunkt-verlag, 2007

[Ull09] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, 2009

[VE08] Voigt, Hendrik ; Engels, Gregor: Ein verfeinerter GQM-Ansatz zur Qualitätsbewertung von Software-Modellen. In: *Universität Paderborn, Deutschland* (2008)

[Wal01] Wallmüller, Ernest: *Software-Qualitätsmanagement in der Praxis*. Hanser, 2001

[Wan09] Wanner, Gerhard: Software-Metriken - Einsatzmöglichkeiten, Nutzen und Trends. In: *OOP 2009*, 2009

[Wei07] Weilkiens, Tim: Die Wogen glätten sich. In: *OBJEKTSpektrum* 4 (2007), S. 36–37

[Whe01] Wheeler, David A.: More Than a Gigabuck: Estimating GNU/Linux's Size. 2001. – Forschungsbericht

[Whe09] Wheeler, David A.: *SLOC*. 2009. – `www.dwheeler.com/sloc`

[Wil00] Will, Hermann: *Mini-Handbuch Vortrag und Präsentation*. Beltz Verlag, 2000

[Wol74] Wolverton, Ray W.: The Cost of Developing Large-Scale Software. In: *IEEE Transactions on Computers* 23 (1974), S. 615–636

Sachverzeichnis