



Philipps-Universität Marburg
Fachbereich: Mathematik und Informatik
Diplomand: Pawel Stepień
Mtr. Nr.: 1492195
Leitung: Prof. Dr. Gabriele Taentzer
WS 2009/2010 – SS 2010

Diplomarbeit

Entwicklung eines Eclipse Plugin zu Metrikbasierten Qualitätsanalyse von Softwaremodellen

Zusammenfassung

Die ständig wachsende Nachfrage nach qualitativ immer hochwertigeren Softwarelösungen führte in den letzten Jahren zu vielen Untersuchungen der Grundlagen für hohe Softwarequalität.

Die Untersuchungen zeigen, dass die Qualität des Endproduktes wesentlich von der Qualität der Zwischenprodukte des Entwicklungsprozesses geprägt wird. Vor allem die frühen Entwicklungsphasen spielen hier eine große Rolle.

Die Modellgetriebene Softwareentwicklung ist eine breit verbreitete Vorgehensweise für Softwareentwicklung. Hier legen die Modelle eine Grundlage für alle späteren Entwicklungsschritte dar. Aus diesem Grund ist es sehr wichtig, dass die in den frühen Entwicklungsphasen entstehenden Modelle eine hohe Qualität aufweisen.

Metriken eignen sich dazu, Softwaremodelle aus den früheren Entwicklungsphasen zu untersuchen und bieten damit eine Möglichkeit, die Qualität der Softwaremodelle effizient zu messen, bewerten und eventuell verbessern.

Die meisten, von den sich zur Zeit auf dem Markt befindenden Entwicklungsumgebungen unterstützen leider keine Metrikenanalysen von Modellen. Die alleinstehenden Applikationen konnten sich noch nicht durchsetzen und werden selten benutzt.

In dieser Arbeit wird eine Erweiterung (Plugin) für das Programmierwerkzeug: Eclipse entwickelt. Das Plugin soll Eclipse um eine benutzerfreundliche Umgebung für Qualitätsanalyse von Softwaremodellen anhand von Metriken erweitern.

Schlüsselwörter: Modellgetriebene Software-Entwicklung, Qualitätsanalyse, Metrik, Eclipse Plugin, Eclipse Modelling Framework, EMF Metrics

Inhaltsverzeichnis:

1. Einführung	4
2. Problemanalyse	5
2.1. Softwarequalität	5
2.2. Modellqualität	6
2.3. Modellmetriken mit EMF Henshin	7
3. Planung	9
4. Anforderungsanalyse	11
4.1. Vorgehensweise	11
4.2. Lastenheft	12
4.3. Anwendungsfalldiagramm	21
4.4. Aktivitäten	23
4.5. Anwendungsfälle	27
4.6. Zusammenfassung	40
5. Entwurf	41
5.1. Einführung	42
5.2. Architektur	43
5.3. Klassenstruktur	47
6. Implementierung	54
6.1. Einführung	54
6.2. Klassen	55
6.3. Sequenzdiagramme	75
7. Benutzerhandbuch	83
7.1 Metriken konfigurieren	83
7.2 Metriken berechnen	85
7.3. Metriken spezifizieren	89
8. Erweiterungsvorschläge	96
9. Ausblick	97
10. Abbildungsverzeichnis	98
11. Literatur	99

1. Einführung

In dieser Arbeit wird die Entwicklung von dem Plugin „EMF Metrics“ vorgestellt. Es handelt sich um ein Eclipse Plugin zu Qualitätsanalyse von Softwaremodellen anhand von Modellmetriken.

Es wird zunächst die Problematik von Softwarequalität und Qualitätsanalysewerkzeugen diskutiert. Anschließend wird das Graphtransformationssystem „EMF Henshin“ als Werkzeug für eine Metrikbasierte Softwareanalyse in frühen Entwicklungsphasen vorgestellt.

Als nächstes wird der gesamte Entwicklungsprozess von dem Plugin „EMF Metrics“ ausführlich beschrieben. Diese Beschreibung beginnt mit der Planung für das Projekt. Anschließend wird die Anforderungsanalyse mit allen dabei entstandenen Dokumenten beschrieben. Diese geht über in die Entwurfsphase in der, eine Umsetzung der Anforderungen dargestellt wird. Schließlich wird die Implementation des Systems vorgestellt.

Des Weiteren, enthält diese Ausarbeitung ein Benutzerhandbuch in dem alle Funktionen von „EMF Metrics“ vorgestellt werden.

Abschließend erfolgt eine Analyse der realisierten Ziele und Erweiterungsmöglichkeiten.

2. Problemanalyse

Im folgendem wird die allgemeine Motivation die zur Realisierung dieses Projektes führte geschildert. Ferner werden die dafür verwendeten Technologien erläutert.

Es folgt eine kurze Einführung in die Problematik der Softwarequalität. Direkt im Anschluss wird die Bedeutung von Qualitätsanalyse in frühen Entwicklungsphasen diskutiert. Als nächstes werden Modellmetriken als ein Werkzeug zur Qualitätsanalyse von Softwaremodellen vorgestellt und Möglichkeiten zur Umsetzung dieses Ansatzes analysiert. Abschließend werden die Entwicklungsumgebung Eclipse und das Graphtransformationssystem EMF Henshin als Lösungsansatz vorgestellt.

2.1. Softwarequalität

Hohe Qualität ist seit langem keine hervorstechende Eigenschaft von Software mehr. Vielmehr wurde sie in den letzten Jahren zu einer „muss“ Anforderung, die an aktuelle Projekte gerichtet wird.

Der Begriff der Qualität entspricht einem mehrdimensionalem Konzept, der hauptsächlich durch die folgenden Eigenschaften geprägt wird:¹

- Funktionalität,
- Beständigkeit,
- Bedienbarkeit,
- Effizienz,
- Wartbarkeit,
- Portabilität.

Die Definition dieser Merkmale ist aber nicht ausreichend um gute Qualitätseigenschaften sicherzustellen. Die Eigenschaften können auf verschiedene Weisen interpretiert werden. Demnach können verschiedene, nicht immer eindeutige und vor allem nicht immer objektive Ansätze bezüglich der Qualitätssicherung entstehen. Neben der Definition von Qualitätseigenschaften benötigen Softwareentwickler ein allgemein geltendes, objektives Qualitätsmessungswerkzeug um diesem Problem aus dem Weg zu gehen.

Es gibt zahlreiche Ansätze zur Messung der Qualität von Software. Einen der Ansätze bilden die

¹ Quelle: [GMPG]

sog. Softwaremetriken (kurz Metriken). Dabei handelt es sich um verschiedene mathematische Funktionen die gewisse Eigenschaften von Software auf einen Zahlenwert abbilden.

Metriken bieten eine wohldefinierte und eindeutige Grundlage zur Qualitätsanalyse von Softwaresystemen und haben sich bereits auf dem Feld der Qualitätsanalyse stark bewährt. Sie können während eines Softwareprojektes eingesetzt werden um die Qualitätseigenschaften der erstellenden Software abzubilden. Anhand von den durch Metriken bereitgestellten Ergebnissen können entsprechende Maßnahmen zur Qualitätssicherung vorgenommen werden um so eine hohe Qualität des Endproduktes zu erzielen.

2.2. Modellqualität

Es stellt sich die Frage an welcher Stelle man die zu untersuchenden Softwaresysteme auf Qualität analysieren sollte. Die meisten bisherigen Ansätze, unter anderem auch Metriken, können erst in späten Phasen eines Softwareprojektes eingesetzt werden da sie auf einer Analyse des Quellcodes aufbauen.

Untersuchungen zeigen aber, dass die Entwurfsphase für die Kosten eines Projektes ausschlaggebend ist. Nicht nur wegen eigener Kosten sondern vielmehr weil sie die Kosten aller folgenden Entwicklungsphasen beeinflusst. Es werden zwar nur ca. 5% der gesamten Zeit in einem durchschnittlichen Softwareprojekt für in Entwurf investiert, dafür geht aber bis zu 80% der gesamten Zeit für Korrektur von Entwurfsfehlern verloren. Die Kosten einer Korrektur steigen bis zu 100 mal wenn diese erst in den späten Entwicklungsphasen durchgeführt wird²

Aus der Sicht der Objektorientierten Softwareentwicklung bei der, der modellgetriebene Entwurf eine zentrale Rolle spielt, ist eine Qualitätssicherung in den frühen Phasen von zentraler Bedeutung. Bei Paradigmen wie „Model-Driven Development“³ und „Model-Driven Architecture“⁴, wo die Erstellung von Modellen im Vordergrund steht und die wichtigsten Entscheidungen noch vor der eigentlichen Implementierung getroffen werden, nimmt die Bedeutung einer qualitativ hochwertigen Modellstruktur deutlich zu. Sie bildet eine Grundlage für alle späteren Entwurfsschritte.

Um den bewährte metrikenbasierte Qualitätsanalyse auch in den frühen Phasen der

² Quelle: [Rei]

³ Quelle: [GePiCa]

⁴ Quelle: [OMG02]

Softwareentwicklung anwenden zu können, wurde eine neue Art von Metriken definiert: Modellmetriken. Diese bilden die Qualitätseigenschaften von Softwaremodellen ab und können somit bereits in den Spezifikations- und Entwurfsphasen eines Softwareprojektes angewendet werden.

Durch eine in den frühen Phasen durchgeführte Metrikenbasierte Qualitätsanalyse eines Softwareprojektes können folgende Ergebnisse erzielt werden:⁵

- Schwachstellen des Entwurfes können noch vor der Implementierung entdeckt und somit kosteneffizient behoben werden.
- Entscheidungen zwischen verschiedenen Entwurfsansätzen können auf einer objektiven Basis getroffen werden.
- Die Charakteristik der Qualitätsmerkmale des Endproduktes kann vorhergesehen werden und Ressourcen können dementsprechend verteilt werden

2.3. Modellmetriken mit EMF Henshin

Modellmetriken unterscheiden sich stark von Codemetriken, und brauchen deswegen neuer Technologien für ihre Umsetzung.

Ein Ansatz zur Implementierung von Modellmetriken basiert auf Graphtransformationsregeln. Diese Regeln können zur Definition von Metriken genutzt werden. Die Idee dahinter ist Graphtransformationsregeln für Softwaremodelle zu definieren, diese aber nicht anwenden, sondern nur überprüfen ob sie angewendet werden könnten beziehungsweise wie oft sie in einem bestimmten Modell angewendet werden könnten. Die Anzahl möglicher Anwendungen einer Transformationsregel entspricht dann dem Wert der Metrik die mit dieser Regel definiert worden ist. Da es sich bei solchen Regeln um ein Metriken abbildende Regeln handelt werden dort keine Veränderungen des Zielmodells definiert. Es werden lediglich Eigenschaften auf die das Modell untersucht werden soll definiert.

Das EMF Henshin⁶ Plugin ist ein Programmiergerüst zur Definition und Ausführung von Graphtransformationsregeln und eignet sich zur Definition von Metriken. Das System beinhaltet einen grafischen Editor in dem Transformationsregeln als Modelle definiert werden können.

⁵ Quelle: [GePiCa]

⁶ Für weitere Informationen zum EMF Henshin Projekt sehe [Hen].

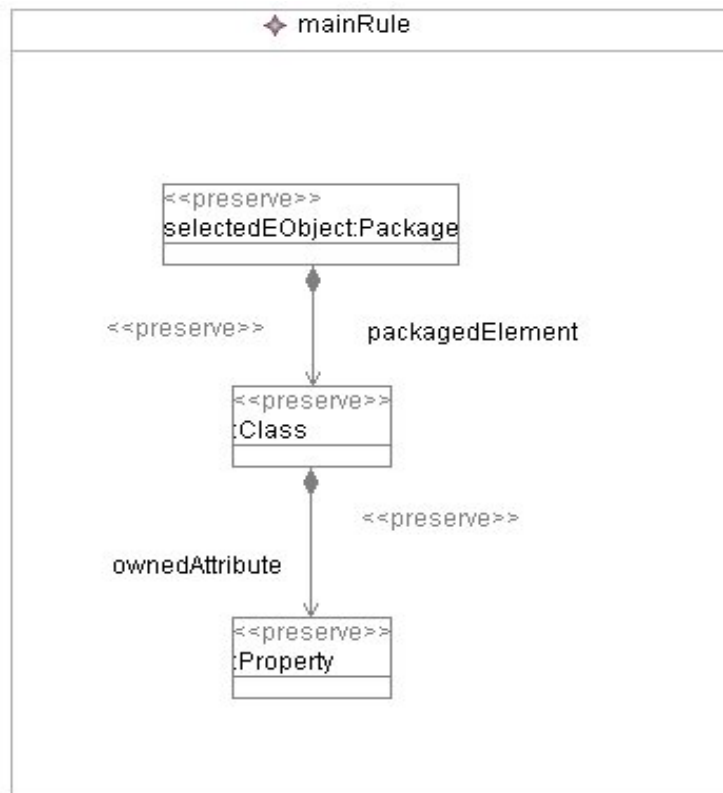


Abbildung 2.1. Henshin-Transformationsregel

Auf Abbildung 2.1 wurde eine Henshin-Transformationsregel dargestellt. Diese Regel entspricht der Modellmetrik: „Number of Attributes“.

Um diese Regel auf einem Bestimmten Element eines Softwaremodells anwenden zu können, muss zunächst eine Assoziation zwischen dem Element und einem Element der Regel aufgebaut werden. Dies wird durch den Parameter: „selectedEObject“ realisiert. Das zu untersuchende Element des Softwaremodells wird der Regel als dieser Parameter übergeben. Die Regel selbst, beinhaltet auch ein Element mit dem Namen „selectedEObject“. Auf diese Weise kann diese Regel im zu untersuchenden Modell verankert werden. Anschließend kann das Modell bezüglich einer potentiellen Anwendung dieser Regel untersucht werden. In der Regel wird eine Beziehung zwischen dem Element mit dem Namen „selectedEObject“ und einer Klasse definiert. Die Beziehung steht für das enthalten sein der Klasse im Package von dem „selectedEObject“-Element.⁷ Des Weiteren wird eine Beziehung zwischen der Klasse und einem Attribut definiert. Diese steht für das einhalten des Attributes durch die Klasse.

Der Wert der Metrik „Number of Attributes“ für ein Package-Objekt eines Softwaremodells entspricht der Anzahl aller möglichen Anwendungen dieser Regel auf dem Objekt.

⁷ Diese Metrik kann dementsprechend nur auf Package-Objekten angewendet werden.

3. Planung

In dieser Entwicklungsphase wurden Entscheidungen bezüglich des verwendeten Vorgehensmodell getroffen.

Die Entwicklung wurde in 8 Phasen aufgeteilt.

1. In der ersten Phase wurde die Aufgabenstellung sowie das in ihr enthaltene Problem analysiert. Es wurde eine theoretische Grundlage bezüglich Modellmetriken und Metrikenbasierter Qualitätsanalyse erarbeitet.
2. In der nächsten Phase wurde die Vorgehensweise bei den nächsten Schritten geplant. Dadurch ist dieses Kapitel entstanden. Es wurde ein Zeitplan (Abb.3.1) erstellt und Deadlines wurden definiert. Des Weiteren wurde eine Risikoanalyse.
3. In der Anforderungsanalyse wurde intensiv mit dem Kunden gearbeitet um seine Wünsche und Erwartungen so genau wie möglich zu erfassen. Anhand von der Anforderungsanalyse wurde eine Spezifikation des zu implementierenden Systems erstellt. Die dabei entstandenen Dokumente werden im Kapitel 4 dieser Arbeit präsentiert.
4. Als nächstes wurde eine Entwurfsphase eingeplant, in der anhand von der Spezifikation die Umsetzung der dort definierten Anforderungen modelliert wurde. Diese Phase bildete eine Brücke zwischen der Systemspezifikation und der Implementierung. Eine Beschreibung der Entwurfsphase befindet sich im Kapitel 5 dieser Arbeit.
5. In der Implementierungsphase wurde anhand von den in der Entwurfsphase erstellten Systemmodelles der Quellcode des Systems sowie Testfälle erstellt. Eine Beschreibung der Entwurfsphase befindet sich im Kapitel 6 dieser Arbeit.
6. Die Testphase wurde fast parallel zur Implementationsphase eingeplant und durchgeführt. Während der Implementation wurden Modultest durchgeführt. Abschließend wurden Integrationstest durchgeführt.
7. Während der Implementierung sind zusätzliche Anforderungen an die Funktionalität des Systems aufgetaucht. Diese wurden berücksichtigt und in die entsprechenden Dokumente der früheren Phasen aufgenommen.
8. Die Dokumentationsphase wurde parallel zu allen anderen durchgeführt und wurde direkt mit der Initiierung des Projektes gestartet. Alle Phasen wurden dokumentiert und im Laufe der Entwicklung entstandene Änderungen wurden in die entsprechenden Dokumente aufgenommen.

Die Dokumentationsphase beinhaltet zusätzlich die Erfassung dieser Ausarbeitung.

Die Planung für dieses Projekt orientiert sich an dem Wasserfallmodell.⁸ Es wurde darauf geachtet dass in jeder Phase Dokumente erstellt werden die in als Eingabe für die nächste Phase dienen können. Die Überlappung der einzelnen Phasen wurde in der Zeitplanung berücksichtigt.

	Januar				Februar				März				April				Mai				Juni			
Problemanalyse																								
Vorgehensweise																								
Anforderungsanalyse																								
Entwurf																								
Implementierung																								
Tests																								
Änderungsanforderung																								
Dokumentation																								

Abbildung 3.1. Zeitplan für das EMF Metrics Projekt

Um den Ereignissen mit hohen Risikofaktoren entgegen zu steuern wurden die folgenden Maßnahmen eingeplant:

- Um den Schäden eines eventuellen Verlustes der Daten zu minimieren, wurde ein SVN System zur Versionsverwaltung eingerichtet
- Ein Zeitpolster von 12 Tagen wurde am ende des Zeitraumes für das Projekt eingeführt um die aufgrund von Erkrankungen oder anderen verhindernden Ereignissen entstandene Verzögerungen neutralisieren zu können.
- Um Koordinations- und Verständnisprobleme zu minimieren wurden tägliche Treffen mit dem Projektbetreuer sowie eine tägliche Kontrolle des SVN-Repository Bestandes eingeplant.

⁸ Das Wasserfallmodell ist ein lineares Vorgehensmodell, bei dem der Entwicklungsprozess in Phasen organisiert wird. Dabei gehen die Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben für die nächste Phase ein. (vgl.[HaMa01] Kapitel 1)

Kapitel 4: Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an das zu entwickelnde System vorgestellt. Es folgt zunächst eine Erläuterung der Vorgehensweise bei der Anforderungsanalyse, anschließend werden die einzelnen dabei entstandenen Dokumente vorgestellt.

4.1. Vorgehensweise

Die Anforderungen wurden in einem direkten Dialog mit dem Kunden erfasst. Dabei ist das Lastenheft entstanden. Es beinhaltet alle Anforderungen an das System aus der Sicht des Kunden. Es wird darin nicht auf die konkrete Realisierung der Anforderungen eingegangen.

Aus der genauen Analyse des Lastenheftes sind das Anwendungsfalldiagramm und die Aktivitätsdiagramme entstanden. Diese Dokumente wurden dem Kunden vorgestellt, gemeinsam besprochen und an einigen Stellen überarbeitet um sicher zu stellen, dass die Anforderungen richtig erfasst worden sind.

Nachdem die Anforderungen erfasst und vom Kunden bestätigt worden sind, wurde eine Spezifikation des Systems erstellt. Die Spezifikation basiert auf den gesammelten Anforderungen stellt sie aber aus der Sicht des Entwicklers dar.

Die einzelnen Anwendungsfälle aus dem Diagramm wurden ausführlich in separaten Dokumenten beschrieben. Dabei wurden schon die ersten Entscheidungen bezüglich der Benutzeroberfläche des Systems getroffen.

Des Weiteren, wurde ein Pflichtenheft erstellt, in dem die Umsetzung der Anforderungen aus dem Lastenheft festgelegt wird. Da in diesem Fall der Kunde selbst ein Softwareentwickler ist, ist es gelungen schon in der Phase der Anforderungssammlung zahlreiche Entscheidungen bezüglich der Spezifikation zu treffen. Aus diesem Grund konnte das Pflichtenheft direkt auf dem Lastenheft aufbauen. Es war nicht nötig ein neues Dokument zu erstellen. Die Inhalte der Spezifikation wurden in das Lastenheft integriert.

4.2. Lastenheft

Das Lastenheft beinhaltet sowohl das in der frühen Phase der Anforderungssammlung entstandene Dokument, wie auch die in der Spezifikationsphase dazugekommenen Informationen. Letztliche werden durch eine kursive Schrift markiert.

L.1. Zielbestimmungen

L.1.1. Produktname: EMF Metrics

L.1.2. Produktbeschreibung

Es soll eine Erweiterung (Plugin) für die Entwicklungsumgebung: Eclipse entwickelt werden.

Das Plugin soll Eclipse um ein Werkzeug zur Qualitätsanalyse von auf dem Eclipse Modelling Framework (EMF) basierenden Softwaremodellen erweitern.

Die Qualitätsanalyse soll durch die Berechnung von Modell-Metriken auf den Softwaremodellen geschehen.

Dem Benutzer sollen Beispiele an vorgefertigten Metriken für UML2-EMF-Modelle zur Verfügung stehen.

Des Weiteren soll der Benutzer die Möglichkeit haben, selber Metriken für EMF basierte Modelle zu spezifizieren. Der Quellcode für die Berechnung der spezifizierten Metriken soll automatisch generiert werden.

Das Plugin soll über eine grafische Benutzeroberfläche (GUI) verfügen.

Die GUI soll benutzerfreundliche Schnittstellen für die folgenden Funktionalitäten bereitstellen:

- Spezifikation von neuen Metriken
- Anzeige der zur Verfügung stehenden Metriken
- Konfiguration einer Auswahl von Metriken für ein gegebenes Projekt
- Berechnung der ausgewählten Metriken auf einem Modell
- Anzeige der Berechnungsergebnisse

Die GUI soll in die Eclipse Entwicklungsumgebung integriert.

Für die Spezifikation von neuen Metriken sollen entsprechende Dialogfenster eingeblendet werden.

Die Anzeige der verfügbaren Metriken sowie die Konfiguration von einer projektspezifischen Auswahl an Metriken soll über das projektabhängige Property-Menu von Eclipse realisiert werden.

Für die Anzeige der Ergebnisse einer Metrikenberechnung wird ein neues View-Element angelegt.

L.1.3. EMF Henshin Kompatibilität :

Für die Spezifizierung neuer Metriken soll Kompatibilität zu dem EMF Modell-Transformations-Werkzeug: „EMF Henshin“ vorhanden sein.

EMF Henshin unterstützt direkte Transformationen von EMF Modellinstanzen wie auch Generierung von Instanzen einer Zielsprache aus einer gegebenen Instanz einer Quellsprache. Dem Benutzer steht für die Spezifizierung der Modelltransformationen sowohl ein baumbasierter (EMF) als auch ein grafischer (GMF) Editor zur Verfügung.

EMF Henshin ist als ein weiteres Eclipse Plug-In realisiert und kann somit direkt in der gleichen Eclipse Instanz wie EMF Metrics aufgerufen werden.

Zur Berechnung von Metriken soll die durch Henshin bereitgestellte Implementation eines Interpreters verwendet werden.

L.2. Produkteinsatz

L.2.1. Benutzergruppe: Softwaremodellierer, Reviewer, Metrikendesigner.

Es wird zwischen zwei Zielgruppen unterschieden.

- Die Softwaremodellierer und Softwarereviewer sollen das Werkzeug zur Berechnung von Metriken benutzen können. Dazu soll eine Auswahl an vorgefertigten Metriken zur Verfügung stehen.
- Die Metrikendesigner sollen mit dem Werkzeug neue Metriken spezifizieren können. Neue Metriken sollen entweder durch die Zusammensetzung der vorhandenen Metriken oder durch die Angabe von Berechnungsregeln spezifiziert werden.

L.2.2. Produkteinsatzbeschreibung

Das Produkt ist als eine Erweiterung von der Eclipse Entwicklungsumgebung konzipiert und kann dementsprechend alleine nicht funktionieren.

Des Weiteren, wird vorausgesetzt dass das EMF Henshin Plugin in die Eclipse Instanz integriert ist.

Das Plugin soll später in ein größeres Modell-Qualitätsanalyse Werkzeugpaket integriert werden.

Es soll aber auch als alleinstehende Eclipse Erweiterung funktionieren können.

Das Plugin soll weltweit genutzt werden können und soll deshalb Englisch als Verkehrssprache benutzen.

L.3. Produktfunktionen

L.3.1. Spezifikation von neuen Metriken für Metamodelle auf EMF.

Die Spezifikation von neuen Metriken soll über grafische Benutzeroberfläche geschehen. Die benötigten Daten sollen durch entsprechende Eingabeformulare erfasst werden.

Für die Spezifizierung von neuen Metriken sollen zwei Szenarien bereit gestellt werden. Eine neue Metrik soll entweder über die Angabe von Berechnungsregeln oder durch Verweise auf bereits vorhandene Metriken spezifiziert werden. Beide Möglichkeiten setzen die Eingabe einer Palette an Basisdaten voraus.

Nachdem alle Informationen für eine neue Metrik erfasst sind, soll die Metrik gespeichert und der Quellcode für ihre Berechnung generiert werden.

L.3.1.1. Eingabe der Basisdaten.

Bei der Spezifikation einer neuen Metrik sollen die folgenden Basisdaten erfasst werden:

- Metriken-Projekt, in das der Code zur Berechnung der neuen Metrik generiert werden soll.
- Name der Metrik.
- Beschreibung der Metrik.
- Metamodell, für das die Metrik angewendet werden soll.
- Kontext, in dem die Metrik angewendet werden soll (z.B. UML.Class)
- Wertebereich, für das Ergebnis der Berechnung der Metrik (z.B. double)

Für die Eingabe dieser Daten soll ein neues Fenster mit einem entsprechendem Formular eingeblendet werden.

L.3.1.2. Beschreibung durch Modelltransformationen.

Mit Hilfe von EMF Henshin sollen Transformationsregeln für die Metriken erstellt werden. Diese sollen dann bei der Spezifikation angegeben und in das System eingebunden werden. Anhand der Transformationsregeln sollen schließlich die Metriken berechnet werden.

Es kann vorkommen, dass eine einzelne Transformationsregel nicht ausreicht, um die gewünschte Metrik zu berechnen. Aus diesem Grund soll es möglich sein, Pakete zu definieren, die aus bis zu drei Transformationen bestehen: eine Transformation, die das Modell für die Berechnung vorbereitet, eine Transformation zur Berechnung der Metrik und eine abschließende Transformation, die das Modell wieder in den Anfangszustand bringt.

Zusätzlich zu den Basisdaten sollen in diesem Szenario die folgenden Informationen erfasst werden:

- Verweise auf die Dateien mit den Modelltransformationen.

L.3.1.3. Zusammensetzung von vorhandenen Spezifikationen.

In diesem Szenario soll zunächst eine Liste mit allen verfügbaren Metriken angezeigt werden. Der Benutzer soll aus dieser Liste die für die Zusammensetzung gewünschten Metriken auswählen können. Des Weiteren soll eine mathematische Operation zur Verbindung der markierten Metriken ausgewählt werden. Die Operation soll ebenso aus einer Liste aller verfügbaren Operationen ausgewählt werden können.

Das Ergebnis der neu spezifizierten Metrik soll aus den Ergebnissen der jeweiligen Metriken unter der Ausführung der mathematischen Operation bestehen.

Zusätzlich zu den Basisdaten sollen in diesem Szenario die folgenden Informationen erfasst werden:

- Verweis auf die erste Metrik
- Verweis auf die zweite Metrik
- Definition der mathematischen Bindeoperation

Für L.3.1.2 und L.3.1.3 sollen weitere Fenster eingeblendet werden in denen die zusätzlichen Daten eingegeben werden können. Für die Metrikenauswahl zur L.3.1.3 soll eine Liste der Verfügbaren Metriken erstellt werden.

L.3.1.4. Speicherung der Metrik.

Nachdem die Metrik erstellt worden ist soll sie auf der Festplatte abgespeichert werden, so dass sie auch nach dem Schließen der Eclipse Instanz beständig bleibt und für spätere Sitzungen zur Verfügung steht.

Es wird gefordert, dass die gespeicherte Metrik beim nächsten Start des Plugin automatisch erkannt wird sofern das Zielprojekt noch vorhanden ist. Dies soll möglichst über interne Strukturen von Eclipse sichergestellt werden.

Das Speichern der neuen Metrik soll durch den erfolgreichen Abschluss des Spezifikationsprozesses automatisch initiiert werden.

Um die neuerstellten Metriken zu speichern soll die Extension-Point Technologie von Eclipse verwendet werden.

L.3.1.5. Generierung des Codes zur Berechnung der Metrik.

Der Code zur Berechnung der neu erstellten Metrik soll bei der Speicherung der Metrik automatisch generiert werden. Dafür sollen keine weiteren Interaktionen mit dem Benutzer erforderlich sein.

Die Datei mit dem Code soll in das aus den Basistaten (L.3.1.1) entnommene Verzeichnis gespeichert werden.

Es soll eine feste Verknüpfung zwischen der Datei mit dem Berechnungscode und den dazugehörigen Henshin Dateien erstellt werden. Es soll aber möglich sein, das Projektverzeichnis in dem sich die beiden Daten befinden zu ändern, ohne dass diese Änderung eine Auswirkung auf die Funktionalität des Codes hat.

Falls nötig, sollen im Zielprojekt neue Klassenabhängigkeiten gesetzt werden, so dass der neu erstellte Code ohne Probleme kompiliert werden kann.

L.3.2. Berechnung der Metriken auf bestimmten Modellen

Die Berechnung der Metriken auf einem bestimmten Modell soll durch die GUI initiiert werden können. Die Liste der zu berechneten Metriken soll vor dem Start der Berechnung über die Benutzeroberfläche auf eine von dem Benutzer definierte Auswahl eingeschränkt werden können.

Nach der erfolgreichen Ausführung der Berechnungen, sollen die jeweiligen Ergebnisse in der GUI angezeigt werden. Es soll möglich sein, die Ergebnisse abzuspeichern.

Des Weiteren soll es möglich sein, die Liste der Ergebnisse zwischen den Berechnungen zu löschen.

Die Berechnung soll über ein Kontextmenü in baumbasierten Modelleditor gesteuert werden. Die Ergebnisse der Berechnungen sollen in einer speziellen View von Eclipse angezeigt werden. Letztere soll über Buttons verfügen mit denen die dort angezeigten Ergebnisse gespeichert oder gelöscht werden können.

L.3.2.1. Laden von Metriken.

Die Metriken sollen bei der Programminitialisierung geladen werden. Dieser Prozess soll automatisch ausgeführt werden. Dafür sollen keine Interaktionen mit dem Benutzer erforderlich sein. Die Liste der bereitstehenden Metriken soll aus allen Metriken bestehen, die in Projekten spezifiziert worden sind, die bei der Programminitialisierung vorhanden waren.

Die Liste der Metriken soll jedes mal bei der Initialisierung einer neuen Sitzung neu geladen werden um sicher zu stellen dass sie aktuell ist. Nach der Terminierung des Plugins soll sie wieder gelöscht werden.

Für das Laden der Metriken soll die Extension-Point Technologie benutzt werden.

L.3.2.2. Anzeigen von geladenen Metriken.

Es soll möglich sein, alle geladenen Metriken über die GUI anzuzeigen. Die Anzeige soll aus den folgenden Feldern bestehen:

- Name
- Beschreibung
- Kontext
- Metamodell

Die Metriken sollen nach Metamodellen gruppiert werden. Des Weiteren soll eine Untergruppierung nach dem Kontext der Metriken erfolgen.

Die Anzeige soll über das projektabhängige Property-Menu von Eclipse realisiert werden.

L.3.2.3. Metriken für Berechnung konfigurieren

In der Metrik-Anzeige (L.3.2.2) soll es möglich sein, die gewünschten Metriken zu markieren, und die so entstandene Auswahl zu speichern. Es soll möglich sein, für jedes Projekt eine andere Auswahl von Metriken zu haben.

Beim öffnen der Metrik-Anzeige (L.3.2.2) für ein bestimmtes Projekt sollen in der Liste aller verfügbaren Metriken die ausgewählten Metriken markiert werden.

Es soll auch möglich sein, diese Markierungen über die Metrik-Anzeige (L.3.2.2) wieder zu entfernen.

Die Anzeige aus L.3.2.2 soll ein zusätzliches Feld für die Konfiguration besitzen. Die Konfiguration soll ebenfalls über das in L.3.2.2 spezifizierte Property-Menu erstellt werden können.

L.3.2.4. Metriken berechnen.

Die Berechnung der Metriken soll aus der GUI gestartet werden können. Nachdem der Benutzer ein konkretes Objekt eines Modells ausgewählt hat, soll er die Möglichkeit haben, Metriken auf dem Objekt zu berechnen.

Die Liste der Metriken für die Berechnung soll anhand der in L.3.2.3 ausgewählten Metriken erstellt werden. Die zu berechnenden Metriken sollen jedoch alle dem Metamodell wie auch dem Kontext des ausgewählten Objektes entsprechen. Die Liste aus L.3.2.3 soll für jedes ausgewählte Objekt entsprechend gefiltert werden.

Es sollen nur diejenigen Metriken berechnet werden, die für das richtige Metamodell spezifiziert worden sind und die für den Kontext des Objektes anwendbar sind.

L.3.2.5. Berechnungsergebnisse anzeigen.

Die Ergebnisse der Berechnungen aus L.3.2.4 sollen in der GUI angezeigt werden. Die Anzeige soll aus den folgenden Feldern bestehen:

- Datum und genaue Uhrzeit der Berechnung
- Der konkrete Kontext in dem die Metrik berechnet worden ist.
- Name der Metrik die berechnet worden ist
- Beschreibung der Metrik.
- Ergebnis der Berechnung

Für weitere Berechnungen sollen die Ergebnisse zu dieser Liste hinzugefügt werden. Dabei sollen die schon vorhandenen Einträge nicht gelöscht werden.

Es soll möglich sein, zwischen einzelnen Berechnungen die Liste der Ergebnisse zu löschen.

Der Ergebnisse der Metrikenberechnungen sollen in dem für sie vorgesehenen View-Element angezeigt werden.

L.3.2.6. Berechnungsergebnisse abspeichern.

Es soll möglich sein, die gesamten Berechnungsergebnisse zu speichern. Der Inhalt der Anzeige aus L.3.2.5 soll auf der Festplatte gespeichert werden können. Dabei sollen Daten aus allen in L.3.2.5 angegebenen Feldern gespeichert werden.

Der Befehl für die Speicherung soll aus der GUI zugänglich sein.

Die Ergebnisse sollen in eine separate Datei, gespeichert werden können. Die Datei soll über ein gut erweiterbares und möglichst unkompliziertes Format verfügen.

L.4. Produktdaten

L.4.1. Die Metriken sollen separat - unabhängig vom Speicherort des EMF Metrics Plugins - gespeichert werden. Sie sollen bei der Plugin Initialisierung leicht geladen werden können.

L.4.2. Die Dateien mit den Berechnungscodes sowie die Henshin Transformationsdateien sollen mit einander fest verknüpft und in gleichem Zielprojekt gespeichert werden.

L.4.3. Die Liste aller verfügbaren Metriken soll nur vorübergehend für jede Sitzung gespeichert werden.

L.4.4. Die projektspezifische Auswahl der Metriken (L.4.2.3) soll in einer separaten Datei ebenfalls in gleichem Projekt gespeichert werden.

L.4.5. Die Berechnungsergebnisse sollen extern gespeichert werden können. Es muss in einer Datei im Textformat gespeichert werden können.

L.5. Zusatzanforderungen

L.5.1. Produktleistungen

L.5.1.1 Benutzerfreundlichkeit: Die Funktionalitäten sollen in der GUI leicht und möglichst intuitiv zu finden sein.

L.5.1.2. Akkumulation: Bei fehlererzeugenden Eingaben soll der Benutzer als Fehlermeldung eine Auflistung aller eingegebenen Fehler.

L.5.1.3. Toleranz: Bei fehlerhaften Eingaben muss der Benutzer die Möglichkeit haben, eine Korrektur der Eingabedaten vorzunehmen, ohne Eingaben wiederholt eingeben zu müssen.

L.5.1.4. Konsistenz: Alle gespeicherten Daten sollen zwischen verschiedenen Sitzungen unverändert bleiben.

L.5.1.5. Korrektheit: Alle spezifizierten Metriken sollen den Spezifikationen entsprechende Werte liefern.

L.5.2. Qualitätsanforderungen

L.6.1.1. Es wird ein besonders großer Wert auf ausführliche Dokumentation gelegt.

L.6.1.2. Es wird eine Modellbasierte Vorgehensweise erfordert.

L.6.1.3. Gute Erweiterbarkeit sollte gewährleistet sein.

L.6. Ergänzung

L.6.1. Realisierung

Als Programmiersprache wird Java verwendet.

Das Plug-In soll in die Eclipse Galileo modeling version eingebunden werden.

Als Modell-Transformations-Werkzeug soll EMF Henshin benutzt werden.

Innerhalb von 6 Monaten soll eine lauffähige beta Version des Plugins entstehen.

L.6.2. Ausblick auf die nächste Version

Dieses System ist hier nur mit den Grundanforderungen angegeben und wird bei Erfolg mit weiteren Funktionalitäten versehen:

- Es werden weitere Metriken spezifiziert und zu der integrierten Liste hinzugefügt.
- Die GUI wird erweitert
- Es wird auf die Ergebnisdatei zugegriffen und der Format wird erweitert
- Das Werkzeug wird in ein größeres Projekt integriert

4.3. Anwendungsfalldiagramm

In dem folgenden Anwendungsfalldiagramm (Abbildung 4.1) werden alle aus dem Lastenheft resultierenden Aktionen des Systems modelliert. Bei der Erstellung des Anwendungsfalldiagrammes wurde darauf geachtet, dass die einzelnen Anwendungsfälle eine optimale Größe bezüglich der realisierten Aufgaben besitzen .

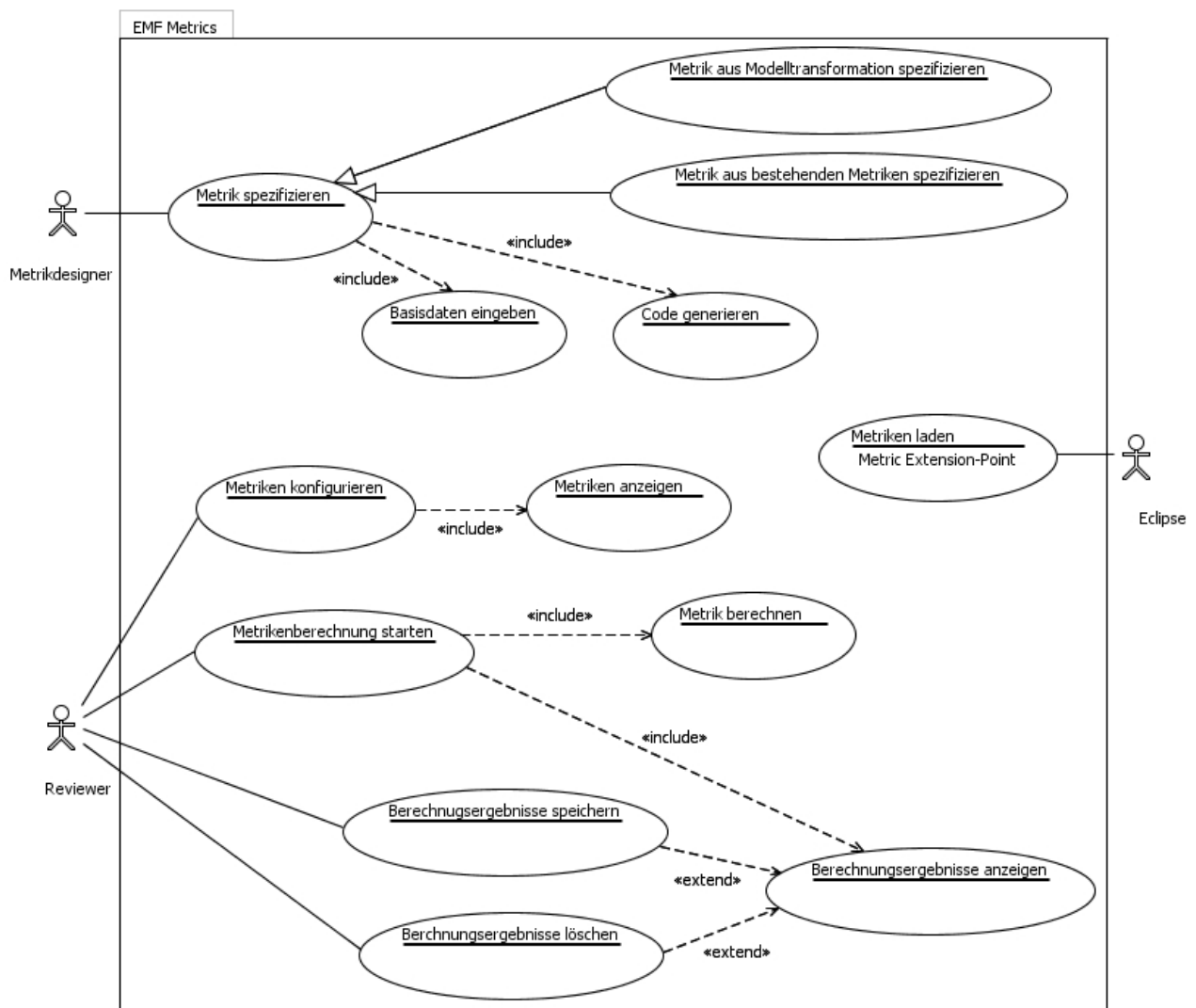


Abbildung 4.1. Anwendungsfalldiagramm: EMF Metrics

Der Aufbau des Diagrammes entspricht den im Lastenheft definierten zwei Hauptfunktionalitäten:

- Spezifikation von neuen Metriken-Projekt (Lastenheft 3.1)
- Berechnung der Metriken auf bestimmten Modellen (Lastenheft: 3.2)

In dem Diagramm wurde die Unterteilung der Benutzer aus dem Lastenheft aufgegriffen. Bei dem Systembenutzer kann es sich entweder um einen Metrikdesigner handeln, der neue Metriken spezifiziert, oder um einen Reviewer, der die schon definierten Metriken dazu nutzt um die Qualität eines Softwaremodells zu untersuchen. Zusätzlich wird Eclipse als Akteur modelliert, da das Laden von verfügbaren Metriken automatisch bei der Initialisierung des Systems geschieht und von Eclipse übernommen wird (Anwendungsfall: „Metriken laden“). Hierzu soll die Eclipse Extension-Point Technologie verwendet werden.

Für die Spezifikation von neuen Metriken gibt es zwei verschiedene Möglichkeiten. Diese beinhalten jeweils die Eingabe von Basisdaten (Anwendungsfall: „Basisdaten eingeben“) und resultieren jeweils mit der Generierung des Metriken Quellcodes (Anwendungsfall: „Code generieren“). Um die Gemeinsamkeiten beider Szenarien zu modellieren wurde eine Vererbungsstruktur eingeführt (Anwendungsfall: „Metrik spezifizieren“). Zusätzlich, abhängig von dem Spezifikationsweg für den sich der Benutzer entscheidet (Anwendungsfall: „Metrik aus Modelltransformation spezifizieren“ oder „Metrik aus bestehenden Metriken spezifizieren“), werden weitere Daten gesammelt.

Um eine projektabhängige Auswahl von Metriken zu definieren (Anwendungsfall: „Metriken konfigurieren“), wird zunächst eine Liste aller verfügbaren Metriken angezeigt (Anwendungsfall: „Metriken anzeigen“).

Die Berechnung der konfigurierten Metriken wird durch den Benutzer auf einem konkreten Element des zu untersuchenden Softwaremodells gestartet (Anwendungsfall: „Metrikenberechnung starten“). Für jede in der Konfiguration enthaltene Metrik wird eine separate Berechnung durchgeführt (Anwendungsfall: „Metriken berechnen“). Nachdem alle Metriken berechnet worden sind, werden alle Ergebnisse auf der Benutzeroberfläche angezeigt (Anwendungsfall: „Berechnungsergebnisse anzeigen“). Diese können auf die Festplatte gespeichert (Anwendungsfall: „Berechnungsergebnisse speichern“) oder eventuell von der Anzeige gelöscht werden (Anwendungsfall „Berechnungsergebnisse löschen“).

Die detaillierten Beschreibungen der einzelnen Anwendungsfälle befinden sich im Unterkapitel 4.5.

4.4. Aktivitäten

Die im Lastenheft (4.2) und im Anwendungsfalldiagramm (4.3) definierten Aktivitäten im System wurden auf den folgenden drei Aktivitätsdiagrammen zusammengefasst. Die Aktivitätsdiagramme beschreiben die möglichen Abläufe im Anwendungsfalldiagramm.

4.4.1. Metrik spezifizieren

In der Abbildung 4.2 wird der Ablauf einer Metrikenspezifikation dargestellt (L.3.1). Es beinhaltet die folgenden Anwendungsfälle: „Metrik spezifizieren“, „Metrik aus Modelltransformation spezifizieren“, „Metrik aus bestehenden Metriken spezifizieren“, „Basisdaten eingeben“ und „Code generieren“.

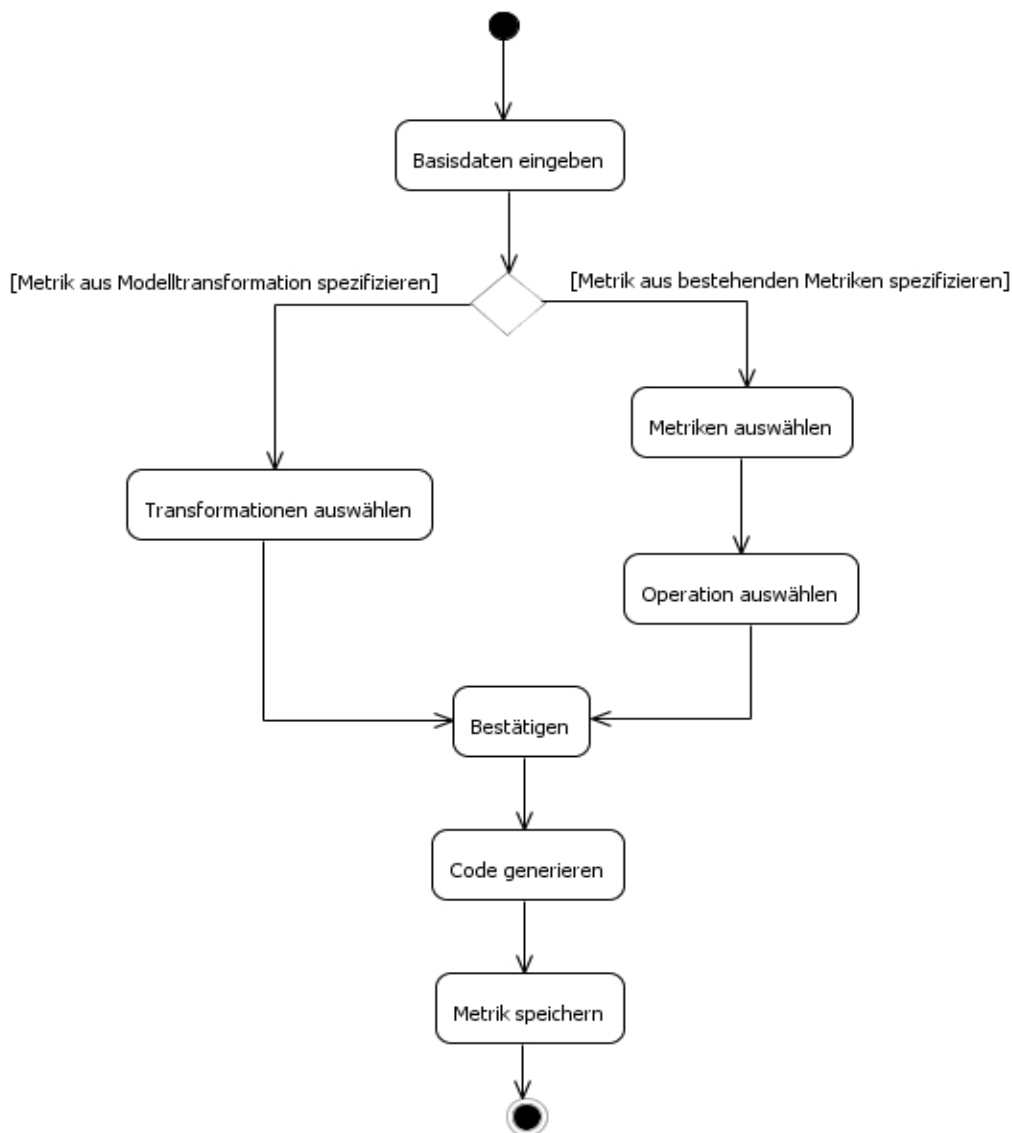


Abbildung 4.2. Aktivitätsdiagramm: Metrik spezifizieren

Die Spezifikation beginnt mit der Eingabe der benötigten Basisdaten (L.3.1.1). Nachdem alle Basisdaten eingegeben wurden, hat der Benutzer die Wahl zwischen einer Spezifikation durch die Angabe von Modelltransformationen (L.3.1.2) oder durch die Zusammensetzung von bereits vorhandenen Metriken (L.3.1.3).

Wird eine Spezifikation durch Modelltransformationen gewählt, erfolgt anschließend die Auswahl von Henshin Transformations-Dateien. Im Fall einer Spezifikation aus bestehenden Metriken, wird der Benutzer gebeten, beide Metriken sowie eine verknüpfende Operation auszuwählen. Beide Szenarien werden mit einer Bestätigung beendet. Der weitere Teil erfolgt automatisch. Zunächst wird der Quellcode der neuen Metrik generiert (L.3.1.4). Nach der erfolgreichen Generierung wird die Metrik gespeichert.

Die einzelnen Aktionen aus den jeweiligen Spezifikationsszenarien sind in den auswahlspezifischen Anwendungsfällen enthalten.

4.4.2. Metriken konfigurieren

In der Abbildung 4.3 wird der Ablauf einer Metrikenkonfiguration dargestellt (L.3.2.3; L.3.2.2). Es beinhaltet die Anwendungsfälle: „Metriken anzeigen“ und „Metriken konfigurieren“.

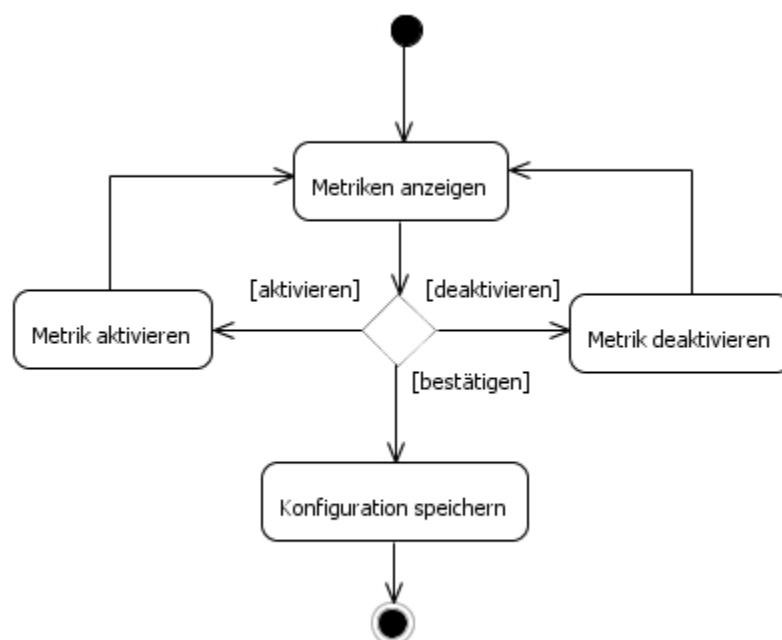


Abbildung 4.3. Aktivitätsdiagramm: Metriken konfigurieren

Die Konfiguration einer projektspezifischen Metrikenauswahl beginnt mit der Anzeige aller geladenen Metriken. Nachdem die Liste aller verfügbaren Metriken angezeigt wurde, kann der

Benutzer jede beliebige Metrik aktivieren oder deaktivieren. Auf diese Weise wird die projektspezifische Konfiguration von Metriken erstellt. Nachdem alle gewünschten Metriken ausgewählt worden sind, kann der Benutzer seine Auswahl bestätigen. Nach der Bestätigung wird die Konfiguration gespeichert.

4.4.3. Metriken berechnen

In der Abbildung 4.3 wurde der Ablauf einer Metrikenberechnung dargestellt (L.3.2.4; L.3.2.5; L.3.2.6). Es beinhaltet die folgenden Anwendungsfälle: „Metrikenberechnung starten“, „Metrik berechnen“, „Berechnungsergebnisse anzeigen“, „Berechnungsergebnisse speichern“, „Berechnungsergebnisse löschen“.

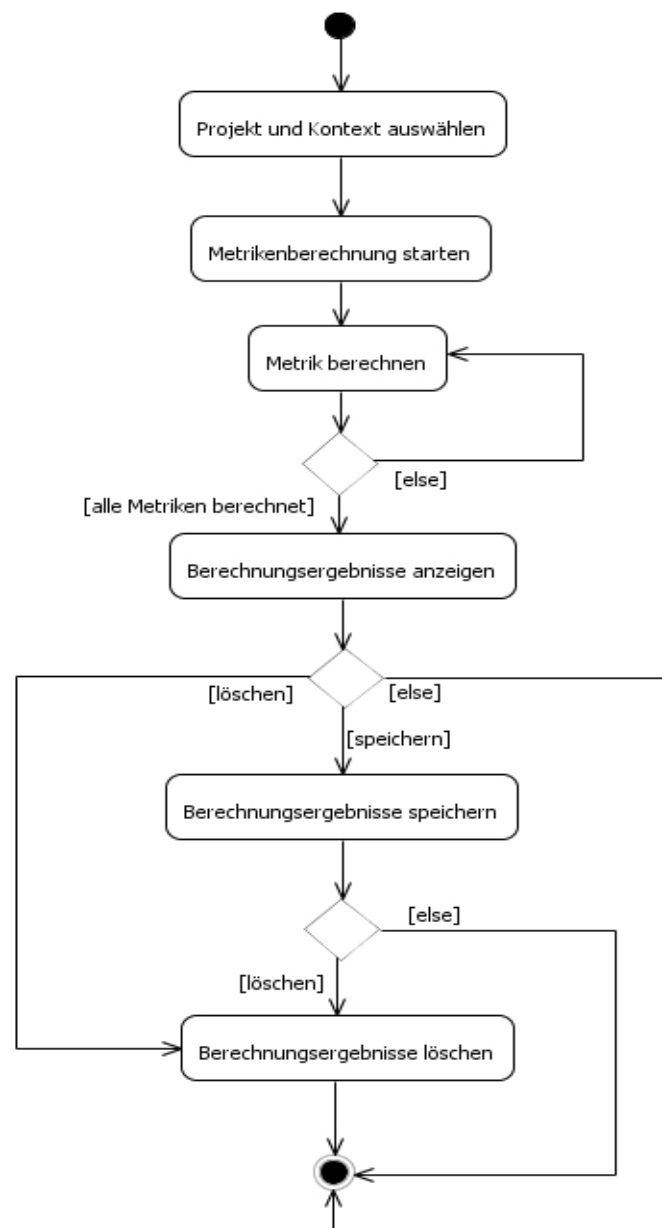


Abbildung 4.4. Aktivitätsdiagramm: Metriken berechnen

Die Berechnung der Metriken wird durch den Systembenutzer initialisiert. Um eine Berechnung zu starten wählt der Benutzer zunächst das Projekt mit dem zu untersuchenden Softwaremodell. Als nächstes muss ein konkretes Element des Softwaremodells ausgewählt werden. Auf diesem Kontext kann die Berechnung der projektspezifischen Metrikenauswahl gestartet werden.

Die Liste der zu berechnenden Metriken ergibt sich dann aus allen für das Projekt konfigurierten Metriken, die dem Kontext des ausgewählten Elementes entsprechen (z.B. `uml.Package` falls es sich um ein Package-Element mit dem UML als Metamodell handelt).

In der Metrikenberechnung werden alle Metriken aus dieser Liste der Reihe nach berechnet (L.3.2.4) Sobald alle Metriken berechnet worden sind, werden die Berechnungsergebnisse auf der Benutzeroberfläche angezeigt (L.3.2.5).

Es besteht die Möglichkeit, die Ergebnisse in eine Datei zu speichern (L.3.2.6) oder die komplette Liste der Ergebnisse aus der Anzeige zu löschen.

4.5. Anwendungsfälle

Die einzelnen Anwendungsfälle wurden bei der Spezifikation des Systems erstellt. Als Ausgangsdokumente dienen der Anwendungsfalldiagramm, die Aktivitätsdiagramme sowie das ebenfalls in der Spezifikationsphase überarbeitete Lastenheft. Die Anwendungsfälle beschreiben die genauen Abläufe in den einzelnen Teile des Anwendungsfalldiagrammes sowie die Beziehungen dazwischen. Des Weiteren beinhalten sie die ersten Entscheidungen bezüglich der Umsetzung der Benutzeroberfläche.

4.5.1. Anwendungsfall: Metriken laden

Beschreibung:

Die Metriken sollen über ein Extension Point angebunden werden. Dieser soll bei der Programminitialisierung aufgerufen werden.

Die Metriken werden separat (unabhängig vom Speicherort des Metrik-Plugins) gespeichert. Sie werden automatisch bei der Plugin Initialisierung über ein Extension-Point geladen

Aktorenbeschreibung:

Eclipse

Vorbedingungen:

Das EMF Metrics Plug-In wird gestartet

Grundverlauf:

1. Eclipse mit installiertem EMF Metrics Plugin wird gestartet.
2. Metriken die den Extension-Point bedienen werden geladen.

Nachbedingungen:

Eclipse und das EMF Metrics Plug-In sind initialisiert.

Alle in Eclipse vorhandenen Metriken wurden geladen.

Spezielle Bedingungen:

Jede Metrik soll eine eindeutige ID haben.

Durch die ID werden die Metriken voneinander unterschieden.

Extension Points:

Ein Extension-Point, über den die Metriken aus einer externen Datei geladen werden können.

4.5.2. Anwendungsfall: Metrik spezifizieren

Beschreibung:

Der Benutzer will eine neue Modellmetrik spezifizieren.

Aktorenbeschreibung:

Metrikdesigner

Vorbedingungen:

Der Benutzer hat auf dem folgenden Weg:

New → Other → EMF Metrics → Metric

die Metrikenspezifikation initiiert.

Grundverlauf:

1. *Basisdaten eingeben*

2. Der Benutzer wird danach gefragt wie er die neue Metrik beschreiben will. Dem Benutzer stehen zwei Möglichkeiten zur Verfügung:

 2.1. *Metrik aus Modeltransformation spezifizieren.*

 2.2. *Metrik aus bestehenden Metriken spezifizieren.*

3. *Metrik-Code Generieren*

4. Das Dialogfenster wird geschlossen.

5. Die Metrik wird gespeichert.

6. Die Metrik wird der Metrikenliste hinzugefügt.

Unterverläufe:

Basisdaten eingeben

Metrik aus Modeltransformation spezifizieren

Metrik aus bestehenden Metriken spezifizieren

Metrik-Code Generieren

Nachbedingungen:

Das Dialogfenster wurde geschlossen.

Die Metrik wurde gespeichert.

Die Metrik wurde der Metrikenliste als *nicht aktiv* hinzugefügt.

4.5.3. Anwendungsfall: Basisdaten eingeben

Beschreibung:

Eine neue Metrik wird spezifiziert und der Benutzer wird aufgefordert, die Basisdaten zu der Metrik einzugeben.

Aktorenbeschreibung:

Metrikdesigner

Vorbedingungen:

Der Benutzer hat die Metrikenspezifikation gestartet.

Grundverlauf:

1. Es erscheint ein Dialogfenster mit einem Formular. Das Formular besitzt die folgenden Eingabefelder:

- Name der Metrik.
- Beschreibung.
- Angabe des Metamodells, für das die Metrik angewendet werden kann.
- Angabe des Metriken-Projektes in das der Code generiert werden soll.
Button mit dem man ein Dateidialogfenster öffnet. Im Dateidialogfenster soll der Ort an dem gespeichert werden soll, angezeigt werden.
- Information über Kontext in der Form eines Dropdown-Menüs.
- *Wertebereich (Radiobutton oder Dropdown-Menu)*

2. Der Benutzer bestätigt die Eingabe, indem er den entsprechenden Button klickt.

Nachbedingungen:

Die Daten aus dem Formular wurden erfasst.

Das Dialogfenster mit dem Formular wurde geschlossen.

Das Dialogfenster zur Metrikenerstellung wurde wieder in den Vordergrund geschoben.

4.5.4. Anwendungsfall: Metrik aus Modeltransformation spezifizieren

Beschreibung:

Eine neue Metrik wird anhand von einer Modelltransformation spezifiziert. Die Modelltransformation wird aus einer mit dem EMF Modell-Transformations-Werkzeug: „EMF Henshin“ erstellten Datei eingeladen.

Aktorenbeschreibung:

Metrikdesigner

Vorbedingungen:

Der Benutzer hat im dem „Specify Metric“ Dialogfenster „Henshin“ ausgewählt.

Basisdaten eingeben wurde abgeschlossen.

Grundverlauf:

1. zusätzlich zu den Basisdaten wird der Benutzer aufgefordert einen Verweis auf die Datei mit der Modelltransformation anzugeben. Dies soll über ein weiteres Dateidialogfenster geschehen.
2. Nachdem der Benutzer alle Felder gefüllt hat klickt er auf den Bestätigungsbutton.
3. Die Modelltransformation wird aus der angegebenen Datei geladen.
4. Die Daten aus der Metrikspezifikation werden erfasst.

Unterverläufe:

Basisdaten eingeben

Nachbedingungen:

Die Daten aus der Metrikspezifikation wurden erfasst.

Die Modelltransformation wurde aus der angegebenen Datei erfolgreich geladen.

4.5.5. Anwendungsfall: Metrik aus bestehenden Metriken spezifizieren

Beschreibung:

Eine neue Metrik wird anhand von bestehenden Metriken spezifiziert. Die bestehenden Metriken werden mit einem Operator gebunden.

Aktorenbeschreibung:

Metrikdesigner

Vorbedingungen:

Der Benutzer hat im dem „Specify Metric“ Dialogfenster: „Composite“ ausgewählt.
Basisdaten eingeben wurde abgeschlossen.

Grundverlauf:

1. Nachdem der Benutzer alle Felder gefüllt hat klickt er auf den Bestätigungsbutton.
2. Es erscheint ein weiteres Fenster indem der Benutzer aus einer Liste der verfügbaren Metriken, zwei Metriken auswählen kann.
3. Der Benutzer wählt die Operation, mit der die Metriken gebunden werden.
4. Der Benutzer klickt auf den Bestätigungsbutton.
5. Die Daten aus der Metrikspezifikation werden erfasst.

Nachbedingungen:

Die Daten aus der Metrikspezifikation wurden erfasst

4.5.6. Anwendungsfall: Code Generieren

Beschreibung:

Anhand einer Metrikenspezifikation wird der Code zur Metrikenberechnung automatisch generiert und gespeichert.

Aktorenbeschreibung:

EMF Metrics.

Vorbedingungen:

Eine Metrik wurde spezifiziert.

Grundverlauf:

1. Der Code zur Berechnung der Metrik wird automatisch erstellt.
2. Der erstellte Code wird in eine Datei gespeichert.
3. Ein Eintrag in die plugin.xml Datei des in *Basisdaten eingeben* angegebenen Projektes wird erstellt.

Alternativverlauf:

1. Der Code zur Berechnung der Metrik wird automatisch erstellt.
2. Der erstellte Code wird in eine Datei gespeichert.
3. *(kann Anforderung)* Ein JUnit Testgerüst für den Metrik-Code wird automatisch erstellt.
4. *(kann Anforderung)* Der erstellte Code und die JUnit Tests werden in eine Datei gespeichert.
5. Ein Eintrag in die plugin.xml Datei des in *Basisdaten eingeben* angegebenen Projektes wird erstellt.

Nachbedingungen:

Der Code zur Metrikenberechnung wurde erfolgreich generiert und gespeichert.

Ein Eintrag in die plugin.xml Datei des entsprechenden Projektes wurde erstellt.

(kann) Die JUnit Testgerüste wurden erfolgreich generiert und gespeichert.

4.5.7. Anwendungsfall: Metriken konfigurieren

Beschreibung:

Dem Benutzer wird eine Liste von Verfügbaren Metriken angezeigt. Über diese Liste kann er bestimmte Metriken für die spätere Berechnung auswählen.

Aktorenbeschreibung:

Reviewer.

Vorbedingungen:

Die Metrikenzusammenstellung wurde geladen.

Grundverlauf:

1. *Metriken anzeigen*

2. Der Benutzer wählt die gewünschte Metrik aus und:

 2.1. *aktiviert* sie durch einen Klick auf das entsprechende Feld.

 2.2. *deaktiviert* sie durch einen Klick auf das entsprechende Feld.

Die Schritte 2.1. und 2.2. werden solange wiederholt, bis der Benutzer alle gewünschten Metriken aktiviert hat.

4. Der Benutzer klickt auf den Bestätigungsbutton

5. Das Fenster wird geschlossen.

6. Die Metrikenauswahl wird gespeichert.

Unterverlauf:

Metriken anzeigen

Nachbedingungen:

Die Metrikenauswahl wurde gespeichert.

Das Anzeigefenster wurde geschlossen.

4.5.8. Anwendungsfall: Metriken anzeigen

Beschreibung:

Der Benutzer lässt sich über das Property-Menü des Projektes eine Liste von Verfügbaren Metriken anzeigen. Die Metriken werden nach Metamodellen gruppiert.

Aktorenbeschreibung:

Reviewer.

Vorbedingungen:

Der Benutzer hat das Property-Menü des Projektes geöffnet und auf das Feld: „EMF Metrics“ geklickt.

Grundverlauf:

1. In dem entsprechendem Fenster des Property-Menus des Projektes wird eine Liste von verfügbaren Metriken angezeigt.

Nachbedingungen:

Die Metrikenauswahl wird angezeigt

Spezielle Anforderungen:

Die Metriken in der Liste werden nach Metamodellen gruppiert.

Eine weitere Untergruppierung nach dem Kontext der Metrik.

4.5.9. Anwendungsfall: Metrikenberechnung starten

Beschreibung:

Die ausgewählten Metriken werden für das Projekt berechnet und die Berechnungsergebnisse werden im entsprechendem Fenster angezeigt.

Aktorenbeschreibung:

Reviewer.

Vorbedingungen:

Die Metriken für das Projekt wurden konfiguriert (ausgewählt).

Ein konkretes Element des zu untersuchenden Softwaremodell wurde angeklickt.

Grundverlauf:

1. Der Benutzer klickt in dem Kontextmenü des baumbasierten Modelleditors unter „EMF Metrics“ auf „Calculate“
2. *Metrik berechnen* wird für jede ausgewählte Metrik ausgeführt
3. *Berechnungsergebnis anzeigen*

Unterverläufe:

Metrik berechnen

Berechnungsergebnis anzeigen

Nachbedingungen:

Alle ausgewählten Metriken wurden berechnet

Die Ergebnisse aller Berechnungen wurden angezeigt

4.5.10. Anwendungsfall: Metrik berechnen

Beschreibung:

Eine Metrik wird für das Projekt berechnet.

Aktorenbeschreibung:

EMF Metrics

Vorbedingungen:

Die Metriken für das Projekt wurden konfiguriert (ausgewählt).

Der Benutzer hat die Metrikenberechnung gestartet.

Grundverlauf:

1. Die Metrik wird berechnet.
2. Das Ergebnis der Berechnung wird zwischengespeichert.

Nachbedingungen:

Die Metrik wurde berechnet.

Das Ergebnis wurde für die spätere Anzeige zwischengespeichert.

4.5.11. Anwendungsfall: Berechnungsergebnisse anzeigen

Beschreibung:

Die Ergebnisse der Berechnungen für alle ausgewählten Metriken werden angezeigt.

Aktorenbeschreibung:

EMF Metrics

Vorbedingungen:

Die Metriken für das Projekt wurden konfiguriert (ausgewählt).

Der Benutzer hat die Metrikenberechnung gestartet.

Die Metrikenberechnung wurde für alle ausgewählten Metriken erfolgreich ausgeführt.

Grundverlauf:

1. Für jede berechnete Metrik werden die folgenden Schritte ausgeführt:

- Das zwischengespeicherte Ergebnis wird geladen.
- Das Ergebnis wird zu der Liste der Gesamtergebnisse hinzugefügt.
- Das Ergebnis wird aus dem Zwischenspeicher entfernt

2. Die Liste mit allen Ergebnissen wird angezeigt.

Nachbedingungen:

Die Ergebnisse wurden angezeigt.

Zusatzbedingungen:

Für jede berechnete Metrik befindet sich ein Ergebnis in der Gesamtliste.

4.5.12. Anwendungsfall: Berechnungsergebnisse speichern

Beschreibung:

Die angezeigten Ergebnisse der Berechnungen aller ausgewählten Metriken werden extern gespeichert.

Aktorenbeschreibung:

Reviewer.

Vorbedingungen:

Die Metriken wurden berechnet und die Berechnungsergebnisse wurden angezeigt.

Grundverlauf:

1. Der Benutzer klickt auf den „Save results“ Button.
2. Es erscheint ein Dialogfenster mit einem Dateidialog in dem der Benutzer den Dateinamen und Pfad für die zu speichernde Ergebnisse angeben kann.
3. Die Berechnungsergebnisse werden in der vom Benutzer angegebenen Datei gespeichert.
4. Das Dialogfenster wird geschlossen.

Nachbedingungen:

Die Ergebnisse aller Berechnungen wurden angezeigt.

Das Dateidialogfenster wurde geschlossen.

4.5.13. Anwendungsfall: Berechnungsergebnisse löschen

Beschreibung:

Die angezeigten Ergebnisse der Berechnungen aller ausgewählten Metriken werden gelöscht.

Aktorenbeschreibung:

Reviewer.

Vorbedingungen:

Die Metriken wurden berechnet und die Berechnungsergebnisse wurden angezeigt.

Grundverlauf:

1. Der Benutzer klickt auf den „Clear results“ Button.
3. Die Berechnungsergebnisse werden von der Liste entfernt.

Nachbedingungen:

Die Ergebnisse aller Berechnungen wurden von der Liste entfernt.

4.6. Zusammenfassung

Die Anforderungsanalyse bildet ungefähr ein Drittel des gesamten Projektes. Dies hat verschiedene Gründe. Zum einen überschneidet sich die Anforderungsanalyse stark mit dem Entwurf. Diese Überschneidung ist sehr gut sichtbar in der Spezifikationsphase, wo schon die ersten Entscheidungen bezüglich der Umsetzung des Projektes getroffen wurden. In diesem Projekt wurde diese Überschneidung noch besonders verstärkt, da es sich um ein System handelt, dass eventuell von dem Kunden selbst weiterentwickelt werden soll. Aus diesem Grund bezogen sich die Anforderungen nicht nur auf die Funktionalität des Systems, sondern auch Teilweise auf die Umsetzung und die verwendeten Technologien.

Zum anderen wurde bei diesem Projekt absichtlich ein besonders großer Wert auf eine ausführliche Dokumentation gelegt.

Durch die ausführliche Anforderungsanalyse hat der Kunde die Möglichkeit gehabt sich einen ersten Eindruck von dem entstandenen Produkt verschaffen zu können. Auf diese Weise konnte sichergestellt werden, dass die Anforderungen des Kunden richtig verstanden worden sind.

Durch die gemeinsame Analyse der Anwendungsfälle „gegen“ die Anforderungssammlung im Lastenheft konnte das System bezüglich der Vollständigkeit seiner Funktionalitäten überprüft werden. Dabei hat der Kunde zuerst die verschiedenen Anwendungsfälle untersucht und sie anschließend mit dem Lastenheft verglichen, um zu überprüfen ob alle Anforderungen aus dem Lastenheft durch die Anwendungsfälle realisiert werden.

Die hier vorgestellten Dokumente werden für alle folgenden Entwicklungsschritte benötigt. Ohne die genaue Spezifikation des Systems wäre ein Entwurf zeitintensiv, und fehleranfällig.

Da sich dieses Projekt sich an dem Wasserfallmodell orientiert, ist die Anforderungsanalyse umso wichtiger. Die Entwurfsphase benötigt eine genaue und ausführliche Spezifikation des Systems als Eingabe.

Kapitel 5: Entwurf

Anhand der im Kapitel 4 vorgestellten Dokumenten wurde ein Entwurf des Systems erstellt. In diesem Kapitel werden zunächst die Bestandteile des Entwurfes sowie die Vorgehensweise bei deren Erstellung geschildert. Danach wird das entworfene System anhand der einzelnen Entwurfskomponenten genau beschrieben.

Die Grenze zwischen dem Entwurf und der Implementierung ist unklar. Softwareentwickler wie Niklaus Wirth meinen, der gesamte Entwurf wäre nur ein Teil der Implementierung und jede Entwurfsentscheidung ein Implementierungsschritt. Es gibt auch entgegengesetzte Meinungen wie die von Dick Hamlet und Joe Maybee, dass die Implementierung eigentlich nichts weiteres als der letzte Schritt der Entwurfsverfeinerung ist.⁹

Um ein Kompromiss zwischen den beiden Sichtpunkten zu erzielen, wurde die Grenze wie folgt definiert: die Projektelemente, die einen Softwareentwickler zur Interpretation benötigen und die nicht direkt durch den Rechner ausgeführt werden können wurden dem Entwurf zugeteilt. Alle tieferen Abstraktionsebenen die direkt ausführbare Resultate liefern, gehören zur Implementierung. Die einzige Ausnahme sind die Sequenzdiagramme, die sehr konkrete interne Systemabläufe definieren und deswegen im Kapitel 6 zusammen mit der Beschreibung der Implementierung vorgestellt werden.

⁹ Quelle: [HaMa01] Seite 246-248

5.1. Einführung

Im Gegenteil zur Spezifikation steht beim Entwurf die Umsetzung der Anforderungen und nicht die Anforderungen selbst im Vordergrund. Es wird nicht mehr gefragt *was* erstellt werden soll, sondern vielmehr *wie* es erstellt werden soll. Somit bildet der Entwurf eine Brücke zwischen der Systemspezifikation und der eigentlichen Implementierung. Nach dem Wasserfallmodell besteht das Hauptziel der Entwurfsphase darin, eine Eingabe für die Implementierungsphase zu erstellen. Die Entwurfsdokumentation bildet eine Art Anleitung für die Implementierung.

Der Entwurf wurde auf zwei Abstraktionsebenen durchgeführt. Zuerst ist eine abstrakte Systemarchitektur entstanden. Danach wurde der detaillierte Entwurf in Form eines Klassenmodells mit mehreren Diagrammen erstellt.

Die Systemarchitektur stellt die allgemeine Aufteilung des Systems in seine Komponenten dar. Es werden Schnittstellen zwischen den Hauptteilen, sowie Ein- und Ausgänge des Systems definiert. Dabei wurde darauf geachtet, ein System mit möglichst guter Dekomposition der einzelnen Komponente zu erschaffen.

In dem Klassenmodell werden die statischen Aspekte der Entwurfsperspektive modelliert. Bei der Konzipierung des Klassenmodells wurde ein Iterationsansatz verfolgt. So ist zuerst ein einfaches Diagramm entstanden, in dem nur die Hauptklassen modelliert worden sind. Dann wurden schrittweise alle weiteren benötigten Klassen hinzugefügt.

In dieser Phase wurden auch die allgemeinen Blackbox-Testfälle definiert. Als Basis wurden die Anwendungsfälle aus der Spezifikation¹⁰ verwendet:

¹⁰ Die Anwendungsfälle wurden in Unterkapitel 4.5 vorgestellt.

5.2. Architektur

In Abbildung 5.1 wurde ein abstraktes Schema des Systemaufbaus dargestellt. Es soll den allgemeinen Aufbau des Systems bezüglich seiner Hauptkomponenten schildern. Letztere sollen nach dem „teile und herrsche“-Ansatz¹¹ in den nächsten Schritten iterativ weiter aufgeteilt werden

5.2.1. Überblick

Das EMF Metrics System wird in drei Komponenten aufgeteilt. Die Loader-Komponente ist ausschließlich für das Laden der in verschiedenen Plugins spezifizierten Metriken. Die Calculator-Komponente ermöglicht die Ausführung von Metrikenberechnungen sowie die Erstellung von projektspezifischen Metrikenkonfigurationen. Die Generator-Komponente ist für die Erstellung neu spezifizierter Metriken zuständig. Das System besitzt sechs Dateischnittstellen, die meisten davon unterstützen das XML-Format.

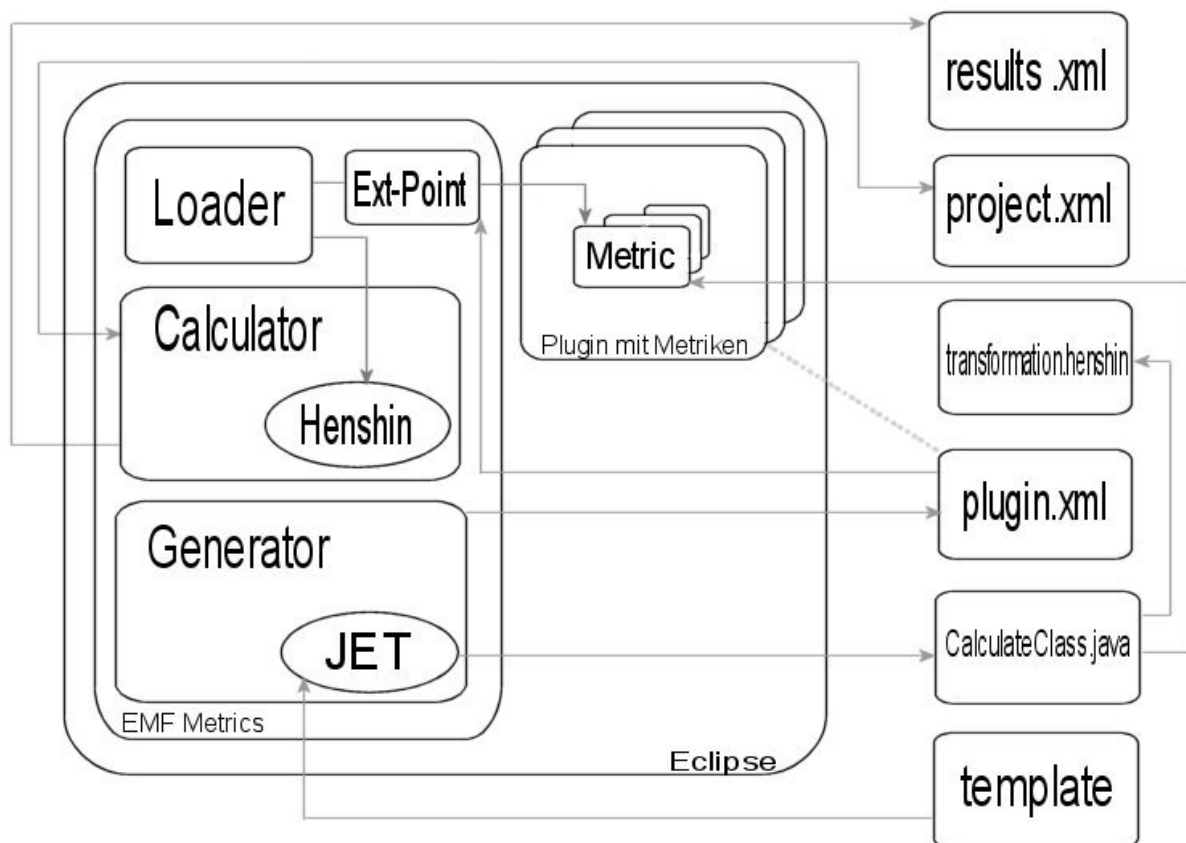


Abbildung 5.1. EMF Metrics Systemarchitektur

11 Engl. „divide and conquer“. Bei einem „teile und herrsche“-Ansatz (engl. „Divide and Conquer“) wird das eigentliche Problem so lange in kleinere und einfachere Teilprobleme zerlegt, bis man diese lösen kann. Anschließend wird aus diesen Teillösungen eine Lösung für das Gesamtproblem (re-)konstruiert.
Quelle: [Wiki01]

5.2.2. Extension-Point

Die Extension für die Metriken wird in EMF Metrics definiert. Alle Projekte in denen Metriken spezifiziert werden, bedienen diese Extension durch die entsprechenden Extension-Points mit den Definitionen der jeweiligen Metriken. Bei der Initialisierung von EMF Metrics werden alle Plugins die die Extension bedienen, automatisch nach Metriken abgefragt.

5.2.3. Plugins mit Metriken

Die Metriken sind in separaten Plugins spezifiziert. Es können mehrere Metriken in einem Plugin definiert werden. Diese Plugins bedienen einen in EMF Metrics definierte Extension und definieren in der Datei „plugin.xml“ ihre Metriken. Die Definitionen in der Datei „plugin.xml“ enthalten sowohl alle Basisdaten der Metriken als auch die Verweise auf die entsprechenden Klassen, in den sich der Code zur Metrikenberechnung befindet.

5.2.4. Loader

Bei der Initialisierung des Systems werden alle in vorhandenen Plugins spezifizierte Metriken über den Extension-Point Mechanismus geladen. Dazu wird die Datei „plugin.xml“ des jeweiligen Plugin untersucht. Dort befinden sich die Definitionen aller für das angegebene Plugin spezifizierten Metriken. So kann der Loader über die Extension-Points auf die Metriken-Plugins zugreifen, und die dort definierten Metriken samt ihrer Java-Klassen in das System laden.

5.2.5. Calculator

Die durch den Loader geladenen Metriken werden an den Calculator weitergegeben. Der Calculator benutzt den Verweis der Metrik auf ihre Java-Klasse und benutzt diese um die Berechnung durchzuführen. Um die Kompatibilität mit Henshin-Graphtransformationen zu gewährleisten, besitzt der Calculator eine Henshin Komponente die für die Behandlung der Henshin Dateien zuständig ist. Bei dem Start einer Metrikenberechnung bezieht sich der Calculator auf eine projektspezifische Konfiguration der Metriken. Diese kann aus einer externen Datei geladen werden um die Beständigkeit der Konfigurationen zwischen verschiedenen Sitzungen zu gewährleisten. Die Ergebnisse der Metrikenberechnungen können in eine Datei exportiert werden.¹²

Erweiterungsmöglichkeit: Die Calculator-Komponente kann problemlos um weitere Metrikenberechnungsansätze erweitert werden, es muss nur eine weitere Komponente hinzugefügt werden, die für eine neue Art der Metrikenberechnung zuständig wäre. Die Definition der Metriken bliebe unverändert, nur die Java-Klasse würde nicht mehr auf die Henshin Komponente zugreifen.¹³

¹² Es werden die results.xml und project.xml Dateien gemeint.

¹³ Stattdessen wurde sie auf die neue – für sie bestimmte Komponente zugreifen.

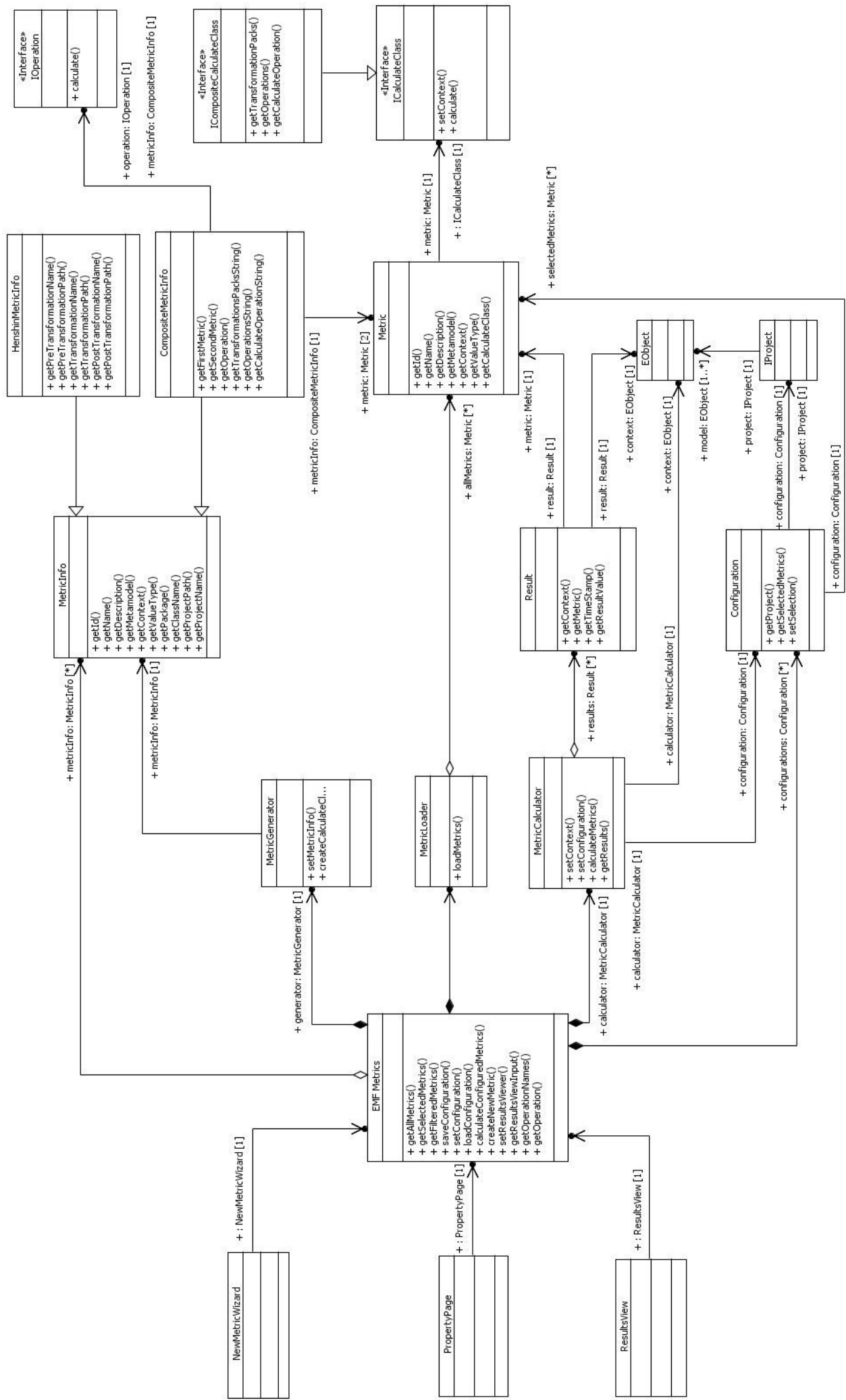
5.2.6. Generator

Bei der Spezifizierung einer neuen Metrik wird die Datei „plugin.xml“ des Projektes in das die Metrik generiert werden soll um den entsprechenden Eintrag erweitert. So wird sichergestellt dass sie bei der nächsten Systeminitialisierung geladen wird.

Der Generator besitzt eine JET Komponente¹⁴ für die Codegenerierung der Java-Klassen. Diese basiert auf einer Auswahl vordefinierter Templates. Die Templates beinhalten ein Grundgerüst der zu generierenden Java-Klassen. Bei jeder neuen Spezifikation wird anhand von den Eingabedaten des Metrikendesigners das entsprechende Template mit Inhalten gefüllt und somit die richtige Java-Klasse einer neuen Metrik generiert.

Erweiterungsmöglichkeit: In der derzeitigen Implementierung enthält die Java-Klasse einen Verweis auf die Henshin Transformationsdatei. Es können aber andere Templates definiert werden, die Metriken auf eine andere Art und Weise berechnen. Die interne Struktur des Generator würde sich in solch einem Fall nicht verändern.

¹⁴ Java Emitter Templates (JET) ist Bestandteil des Eclipse Modelling Framework (EMF) für die generative Erzeugung von Code. (vgl [EMF])



5.3. Klassenstruktur

Das in Abbildung 5.2 vorgestellte Klassenmodell wurde anhand der funktionalen Anforderungen erstellt. Es stellt den konzeptionellen Entwurf der Datenstruktur dar und beinhaltet alle wichtigen Klassen des Systems sowie die Beziehungen dazwischen. Im folgendem wird das Klassendiagramm genau auf seine Struktur untersucht.

5.3.1. Einführung

In dem Diagramm werden die vom System verwendeten Kooperationen modelliert. Mit Kooperationen sind Gruppen von Klassen, Schnittstellen und anderen Elementen gemeint, deren Aufgabe darin besteht, gewisse durch einzelne Elemente nicht darstellbare Systemverhalten zu modellieren.

Beim Entwurf des Klassendiagrammes wurde die Komponentenaufteilung aus 5.2 berücksichtigt. Der obere Teil des Diagrammes entspricht der Generator-Komponente, der untere Teil der Calculator-Komponente, die Loader-Komponente befindet sich in der Mitte (siehe Abbildung 5.3) wobei der mittlere Teil von beiden Komponenten genutzt wird.

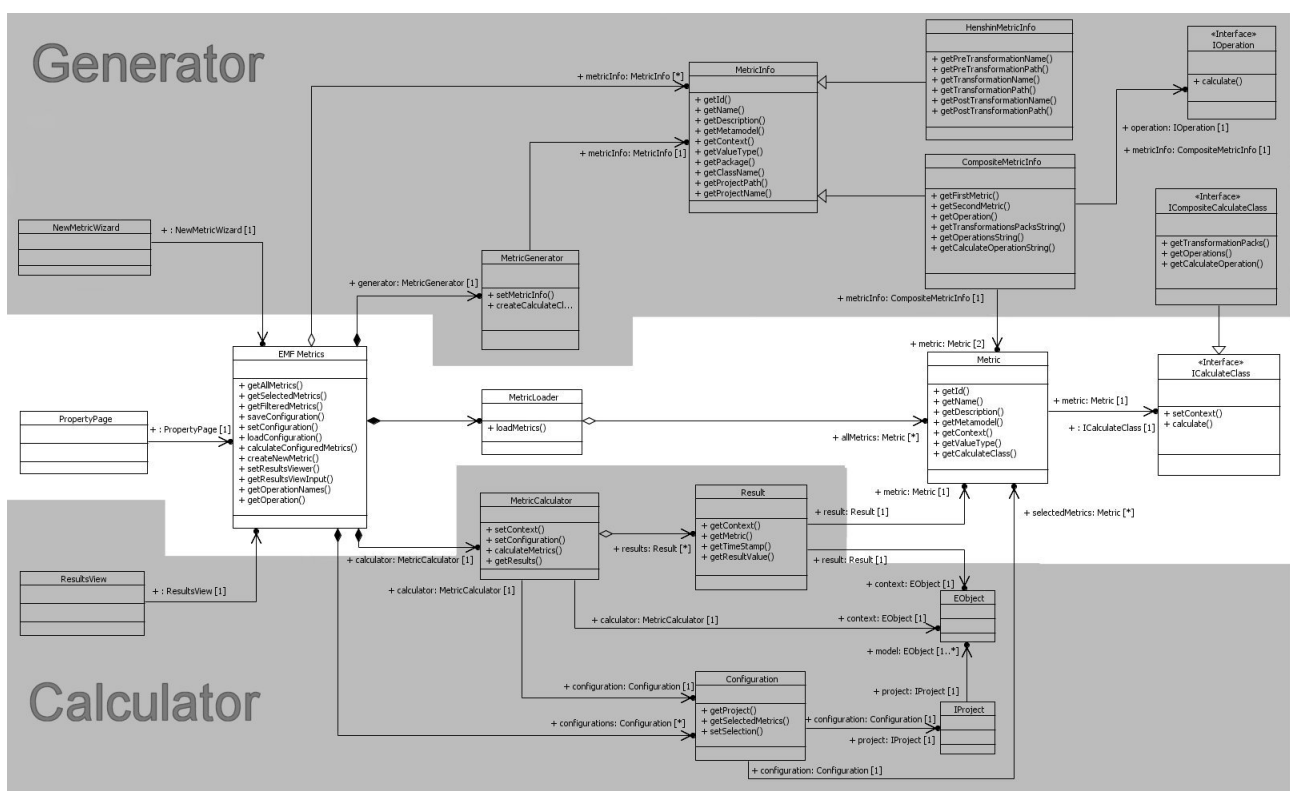


Abbildung 5.3. EMF Metrics Klassendiagramm (Komponenten)

Des Weiteren, wurde hier das Model-View-Controller Konzept (siehe 5.3.2) verwendet. Die interne Systemstruktur wird von der Benutzeroberfläche getrennt. Die Kommunikation wird zwischen diesen sowie verschiedenen internen Elementen wird durch Controller-Klassen bereitgestellt.

Es wurde darauf geachtet, eine starke Datenkapselung zu erzielen. Die meisten Klassenparameter werden als *private* modelliert. Alle Klassen stellen Schnittstellen bereit, die bestimmen, auf welche Weise andere Klassen mit ihnen interagieren.

5.3.2. Model-View-Controller (MVC) Konzept

Ziel des MVC Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.¹⁵

Die allgemeine Funktionalität des Systems wird mit der Controller-Klasse *EMF Metrics* bereitgestellt. Diese bildet den Hauptcontroller des Systems der für die Kommunikation zwischen einzelnen Komponenten sowie auch der Benutzeroberfläche und der internen Systemlogik zuständig ist.

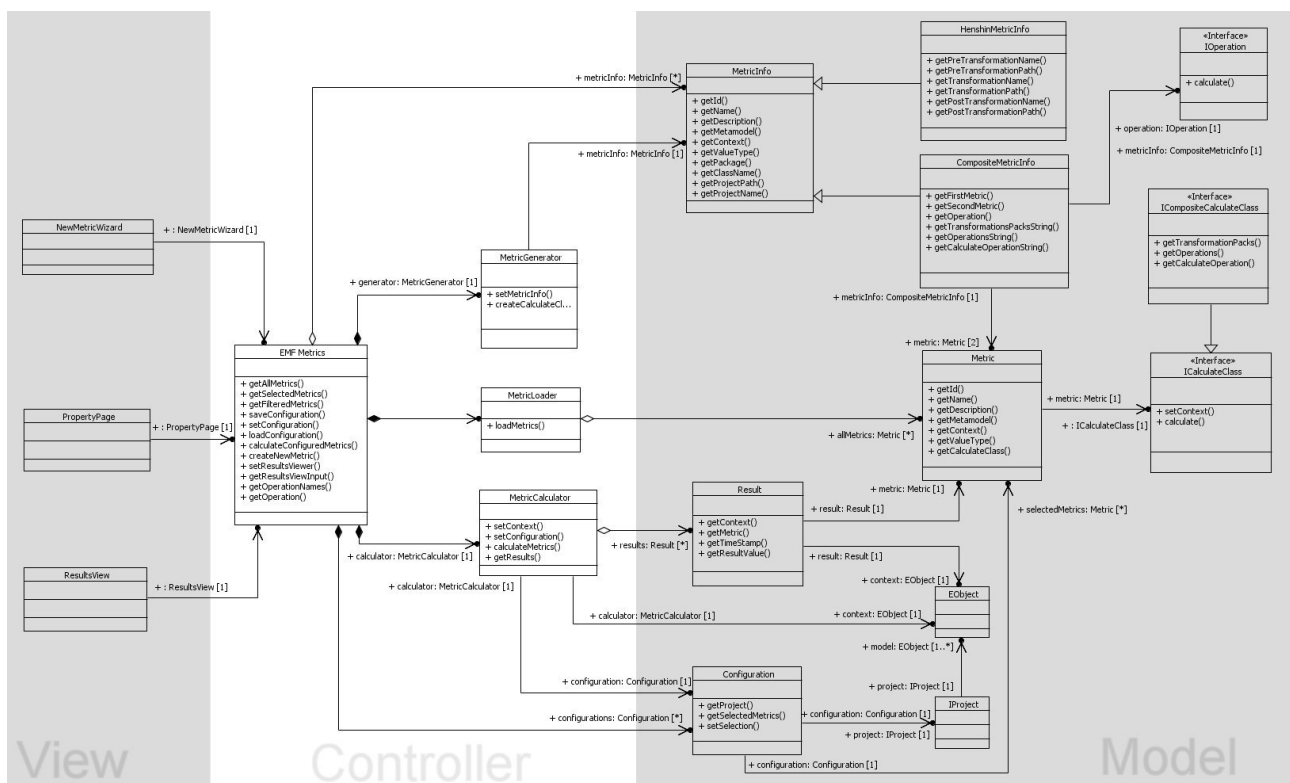


Abbildung 5.4. EMF Metrics Klassendiagramm (MVC)

¹⁵ Das MVC Konzept besteht aus einer Zusammensetzung der Entwurfsmuster: „Strategy“ und „Observer“ (vgl [GHJV95]).

Es gibt drei zusätzliche Hilfscontroller, einen für jede Systemkomponente: *MetricLoader*, *MetricCalculator* und *MetricGenerator*. Sie stellen komponentenspezifische Funktionalitäten bereitgestellt. Je nach Bedarf benutzt der Hauptcontroller diese Klassen.

Alle Befehle der GUI richten sich an den Hauptcontroller. Dieser führt dann die benötigten Operationen aus und aktualisiert die GUI falls nötig. Auf diese Weise wird die interne Systemlogik komplett von der Benutzeroberfläche getrennt.

Die GUI Klassen befinden sich auf der linken Seite des Klassendiagrammes, die Systemlogik auf der rechten Seite und die Controller-Klassen wurden in der Mitte positioniert (siehe Abbildung 5.4).

Erweiterungsmöglichkeit: Durch den MVC Aufbau besitzt das System sehr gute Eigenschaften bezüglich der Austauschbarkeit und Veränderbarkeit. Es ist möglich, die interne Systemstruktur komplett oder teilweise auszutauschen ohne die GUI verändern zu müssen. Die Controller müssten nur an die neuen Schnittstellen zur internen Logik angepasst werden.

Andrerseits haben Veränderungen in der GUI keinen Einfluss auf die interne Systemstruktur. Letztendlich kann der Controller leicht verändert werden (z.B. durch Refactoring), ohne die interne Struktur oder GUI zu beeinflussen.

5.3.3. Gemeinsame Klassen

Die Klasse EMFMetrics bildet den Hauptcontroller des Systems. Sie ist für die Koordination aller Systemprozesse zuständig. Die Klasse Metric beinhaltet alle Basisdaten einer Metrik, sowie einen Verweis auf die Klasse mit Code zur Metrikenberechnung. Beide dieser Klassen tauchen in allen drei Systemkomponenten auf da sie die zentralen Säulen des Systems bilden.

5.3.4. Loader

Die Loader-Komponente (Abbildung 5.5) besteht aus nur einer statischen Klasse: *MetricLoader*. Die Klassen EMFMetrics und Metric sind in allen Komponenten enthalten.

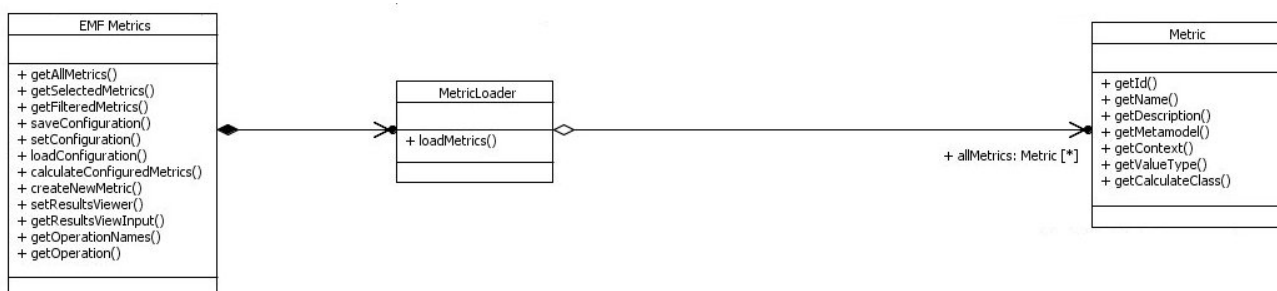


Abbildung 5.5. Loader-Komponente

MetricLoader ist für das Laden der Metriken bei der Systeminitialisierung zuständig. Die Metriken werden über die Extension-Points Technologie geladen. Für jede geladene Metrik wird ein Objekt der Klasse *Metric* erstellt. Um den Ladeprozess nicht wiederholen zu müssen wird die Liste aller geladenen Metriken für jede Sitzung im Hauptcontroller *EMFMetrics* zwischengespeichert.

5.3.5. Calculator

Die in Abbildung 5.7 dargestellte Calculator-Komponente besteht aus insgesamt fünf Klassen: dem Hilfscontroller *MetricCalculator*, der Modellklassen *Configuration* und *Result* sowie zwei externen Klassen *EObject* und einer Instanz von *IProject*.

In der Klasse *EMFMetrics* werden alle projektspezifischen Konfigurationen von Metriken in Form einer Liste von Objekten der Klasse *Configuration* gespeichert.

Für jede einzelne Konfiguration wird ein Objekt der Klasse *Configuration* erstellt und in *EMFMetrics* für die Dauer der Session zwischengespeichert. Jedes Objekt der Klasse *Configuration* besitzt eine Auswahl von Metriken und einen Verweis auf eine Instanz von *IProject*. Letztere entspricht dem Projekt, zu dem die Metrikenkonfiguration gehört.

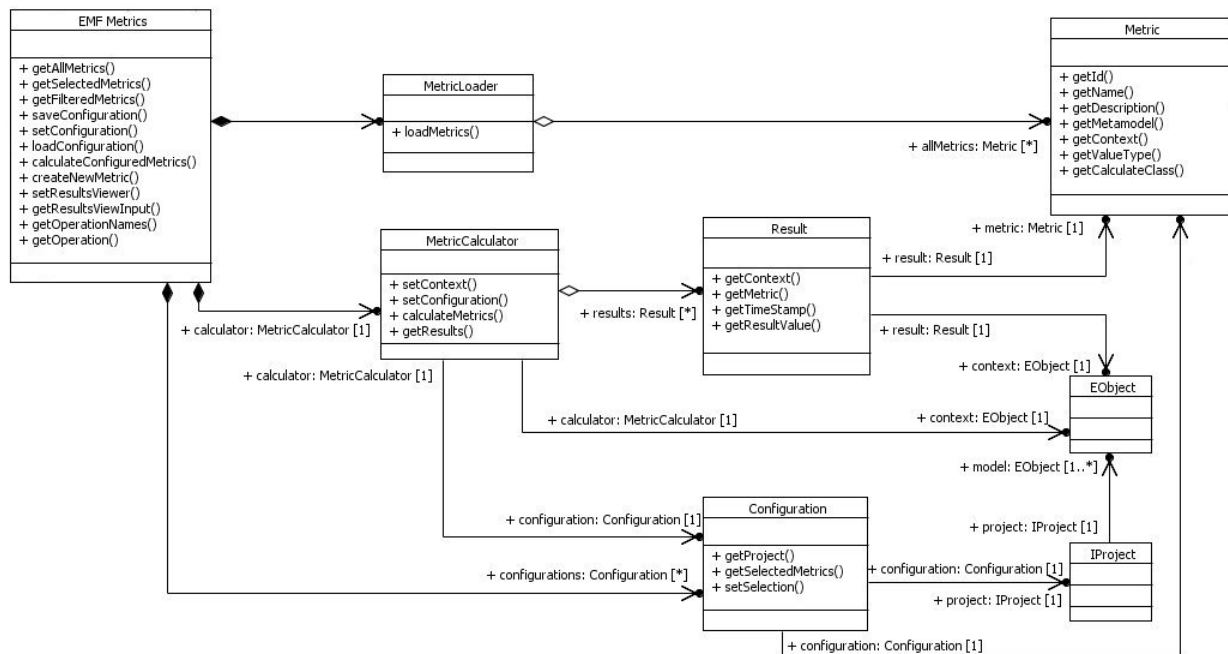


Abbildung 5.6. Calculator-Komponente

Nach dem Ablauf der Sitzung werden alle Konfigurationen in die Datei „projekt.xml“ des jeweiligen Projektes gespeichert.¹⁶ So können die Konfigurationen zwischen einzelnen Sitzungen

¹⁶ Das entspricht der im Unterkapitel 5.2.5 definierten Aktion.

erhalten bleiben. Der Hauptcontroller ist dafür zuständig die gespeicherten Konfigurationen bei Bedarf aus den Dateien „project.xml“ zu laden.

Eine Metrikenberechnung wird in der GUI initialisiert und an den Hauptcontroller weitergegeben. Dieser benutzt die Klasse *MetricCalculator* (Hilfscontroller) um die Berechnung durchzuführen.

Letzterer wird zunächst mit einem Objekt der Klasse *Configuration* initialisiert, derjenigen des aktuellen Projekts entspricht. So wird sichergestellt, dass nur die für das Projekt ausgewählten Metriken berechnet werden.

Danach wird der Kontext, auf dem die Metrikenberechnung stattfinden soll, in der Form eines *EObject* an *MetricCalculator* weitergegeben.

Ein so initialisierter *MetricCalculator* kann die Metrikenberechnungen durchführen und schließlich für jede Berechnung ein Objekt der Klasse *Result* erstellen. Ein *Result* Objekt besitzt Verweise auf die dazugehörige Metrik, den Kontext auf dem sie berechnet wurde, sowie das Ergebnis der Berechnung.

Die Liste von allen *Result* Objekten wird im Hilfscontroller zwischengespeichert und kann vom Hauptcontroller abgerufen werden, um die Ergebnisse an die entsprechende GUI Komponente weiterzuleiten.

5.3.6. Generator

Die in Abbildung 5.6 dargestellte Generator-Komponente besteht aus vier Klassen: dem Hilfscontroller *MetricGenerator*, einer abstrakten Modellklassen *MetricInfo* und zwei konkreten Unterklassen von *MetrikInfo*.

Bei *MetricInfo* wurde das Entwurfsmuster „Composite“ angewendet um eine Gruppe von Objekten in einem Behälter zu sammeln.¹⁷

Die Erstellung der plugin.xml Einträge¹⁸ übernimmt der Hauptcontroller *EMFMetrics*. Die Klasse *MetricGenerator* bildet den Hilfscontroller und ist alleine für das Generieren von neuen Metrik-Klassen zuständig. Die Metrik-Klassen werden anhand von den in *MetrikInfo* gesammelten Daten erstellt.

Die *MetrikInfo* Objekte werden anhand der durch den Benutzer angegebenen Daten aus der GUI erstellt. Je nachdem welche Art der Metrikenspezifizierung der Benutzer gewählt hat, wird ein *HenshinMetricInfo* oder ein *CompositeMetricInfo* Objekt erstellt.

CompositeMetricInfo entspricht dagegen einer Spezifikation durch die Zusammensetzung von

¹⁷ Das Entwurfsmuster wurde in [GHJV95] auf Seite 195 vorgestellt.

¹⁸ Hier werden die im Unterkapitel 5.2.6 genannten Einträge in die plugin.xml Datei gemeint.

vorhandenen Metriken. Die Klasse beinhaltet Verweise auf zwei Metriken, aus denen die neue erstellt werden soll, sowie eine verknüpfende Operation.

Die Parameter für alle Basisdaten der zu erstellenden Metrik werden schon in der Oberklasse deklariert.

Bei der Spezifikation einer neuen Metrik wird ein Objekt der entsprechenden Unterklasse von *MetricInfo* durch den Hauptcontroller erstellt und an die Klasse *MetricGenerator* weitergegeben. *MetricGenerator* behandelt alle Info-Klassen als Instanzen der abstrakten Oberklasse.

Die Unterklassenaufteilung ist lediglich für die Auswahl des richtigen Template für die zu generierende Metrik-Klasse zuständig.

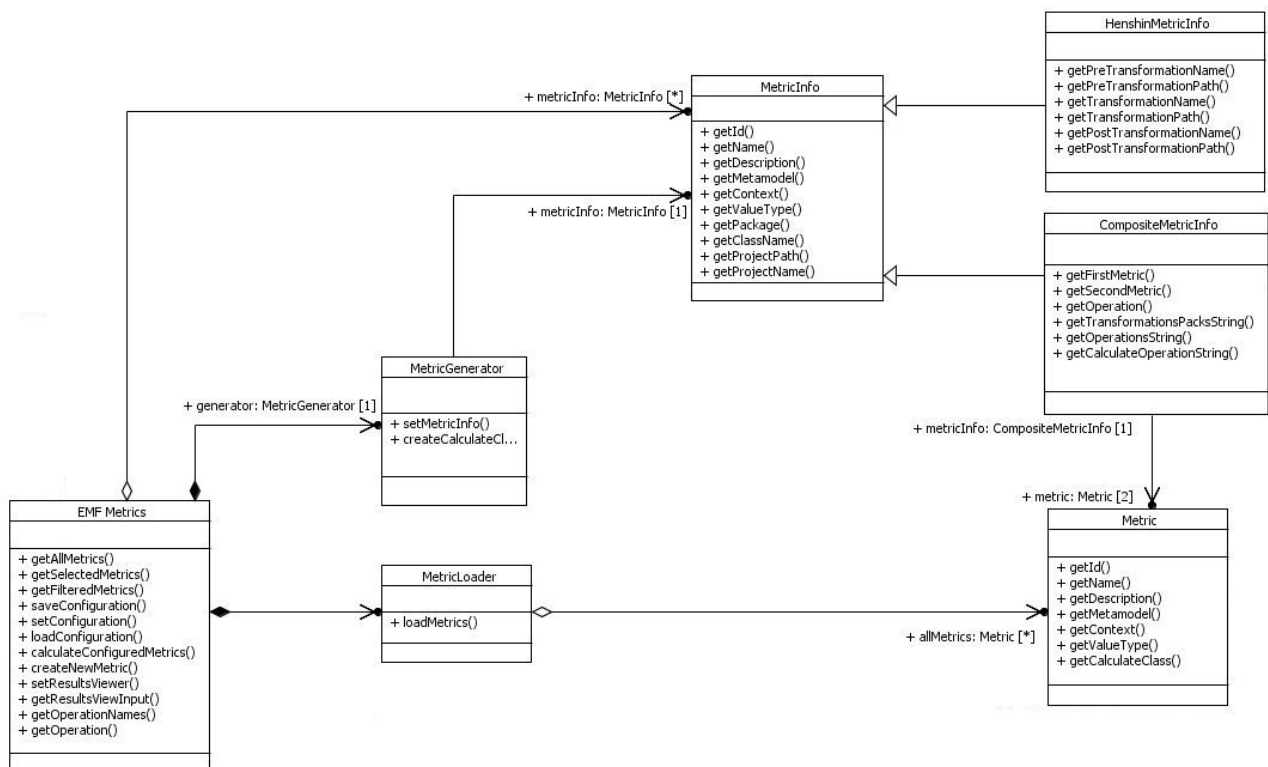


Abbildung 5.7. Generator-Komponente

Erweiterungsmöglichkeit: Dieser Entwurfsansatz ermöglicht eine unkomplizierte Erweiterung des Systems um weitere Arten von Metrikspezifikationen. Es muss nur eine weitere Unterklasse von *MetricInfo* und ein entsprechendes Template hinzugefügt werden.

5.3.7. ICalculateClass, ICompositeCalculateClass

Jede Klasse mit dem Code zur Metrikenberechnung implementiert die *ICalculateClass* Schnittstelle. Auf diese Weise wird sichergestellt, dass die Metrikenberechnungsklassen, die im *MetricCalculator* verwendeten Methoden implementieren.

Die *ICompositeCalculateClass* Schnittstelle ist eine Unterklasse von *ICalculateClass* und bietet zusätzlich die Möglichkeit einer Zusammensetzung von zweier Metrikenberechnungsklassen in denen diese Schnittstelle implementiert wird. Diese Schnittstelle wird für die Spezifikation einer neuen Metrik aus bereits vorhandenen Metriken benötigt. Die Metriken für solche Spezifikation müssen *IcompositeCalculateClass* implementieren.

ICalculateClass wird von der Calculator-Komponente beim Berechnen einer Metrik verwendet. Die Generator-Komponente benutzt nur die *ICompositeCalculateClass* Variante der Schnittstelle bei der Erstellung einer Metrik anhand von anderen Metriken.

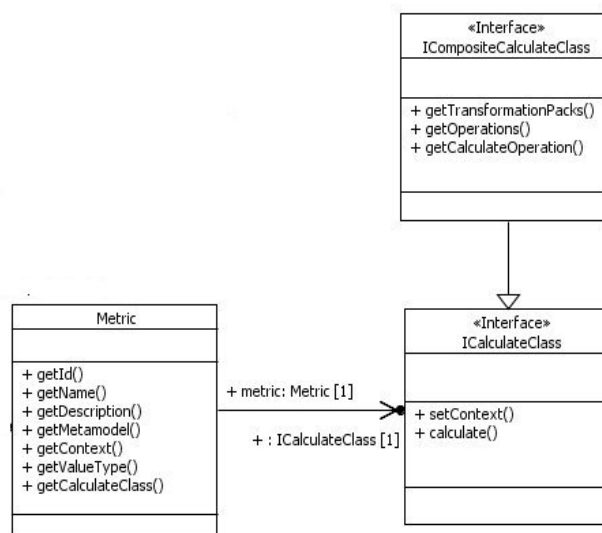


Abbildung 5.8. *ICalculateClass* Schnittstelle

Erweiterungsmöglichkeit: sollten weitere Ansätze für die Zusammenstellung oder Berechnung von Metriken gebraucht werden so können diese weitere Unterklassen von *ICalculateClass* zusammengefasst werden.

Kapitel 6: Implementierung

In diesem Kapitel wird die Umsetzung des Systementwurfes präsentiert. In der Einführung wird zunächst die allgemeine Vorgehensweise bei der Implementierung geschildert. Anschließend werden die einzelnen Klassen mit ihren Attributen und Methoden vorgestellt. Nachdem die einzelnen Klassen präsentiert worden sind, wird die Objektdynamik des Systems anhand von Sequenzdiagrammen vorgestellt. Zum Schluss werden alle weiteren wichtigen Bestandteile des Systems vorgestellt.

6.1. Einführung

Bei der Implementation der im Kapitel 5 entworfenen Klassenstruktur wurde darauf geachtet, dass die einzelnen Klassen übersichtlich gestaltet werden. So wurden zahlreiche Hilfsklassen hinzugefügt. Meistens handelt es sich dabei um statische Klassen die irgendwelche speziellen Funktionalitäten implementieren und statische Methoden für die Hauptklassen bereitstellen.

Die Anzahl an Codezeilen pro Klasse beträgt im Durchschnitt zwischen 50 und 100 Zeilen und es wurde darauf geachtet dass der Quellcode soweit wie möglich selbsterklärend ist.

Es wurden zahlreiche Refactorings durchgeführt um die Qualität der Implementation zu steigern.

Als weitere Qualitätsmaßnahme wurden bei der Implementierung zahlreiche Entwurfsmuster benutzt. Die Controllerklassen wurden als „Singleton“¹⁹ oder als statische Klassen implementiert um sicher zu stellen das immer nur ein Objekt von der jeweiligen Klasse existiert. Des Weiteren wurden an einigen Stellen „Composite“²⁰ Klassen erstellt um Behälter für im System verwendeten Objektsammlungen bereitzustellen.

19 Genaue Beschreibung des "Singleton" Entwurfsmuster kann in [GHJV95] gefunden werden.

20 Genaue Beschreibung des "Composite" Entwurfsmuster kann in [GHJV95] gefunden werden.

6.2. Klassen

Im folgendem wird die interne Implementation der Klassenstruktur präsentiert. Es wurden alle wichtigen Methoden der jeweiligen Klassen vorgestellt. Bei der Auswahl der vorgestellten Methoden wurde der Bezug zur Gesamtfunktionalität des Systems berücksichtigt. So wurden lokale Klassenmethoden nicht ausführlich beschrieben. Diejenigen die keinen Einfluss auf die Systemstruktur haben wurden teilweise komplett übersprungen.

Die Implementation des Systems beinhaltet zahlreiche statischen Hilfsklassen die im Entwurf nicht berücksichtigt wurden weil sie keinen Einfluss auf die Systemlogik haben. Diese wurden nur kurz vorgestellt.²¹

6.2.1. *Activator*

Dies ist die automatisch generierte Klasse des Plugins. Sie enthält Methoden die von Eclipse genutzt werden.

Attribute:

```
public static final String PLUGIN_ID = "de.unimarburg.swt.emf.metrics";
```

Die einzigartige Id des Plugins

```
private static Activator plugin;
```

Die Instanz dieser Klasse die bei bedarf durch die entsprechende getter Methode abgefragt werden kann.

Methoden:

```
public Activator()
```

Der Konstruktor

```
public void start()
```

Mit dieser Methode wird EMF Metrics gestartet.

```
public void stop(BundleContext context)
```

Diese Methode wird beim schließen des Systems aufgerufen.

```
public static Activator getDefault()
```

getter Methode für das plugin Attribut.

²¹ Die GUI Klassen wurden hier nicht vorgestellt. Für einen genauen Einblick in die Klassen in denen die Benutzeroberfläche implementiert wird sollte der Quellcode des Projektes untersucht werden. Informationen zu den dort verwendeten Schnittstellen und dort implementierten Strukturen können [CIRu06] entnommen werden.

2.2. *EMFMetrics*

Diese Klasse bildet den Hauptcontroller des Systems. Sie ist für die Ausführung und Koordination aller Systemprozesse zuständig. Es ist eine statische Klasse.

Erweiterungskompatibilität: Sowohl die GUI-Komponenten als auch die interne Struktur des Systems benutzt diese Klasse um mit einander zu kommunizieren. Sollten neue Komponenten zu dem System hinzugefügt werden, so brauchen sie nur diese Klasse zu verwenden um mit dem Rest des Systems in Verbindung zu treten.

Attribute:

private static LinkedList<Configuration> *configurations*

Eine Liste der zwischengespeicherten Metrikenkonfigurationen für verschiedene Projekte.

private static MetricCalculator *calculator*

Ein Objekt der singleton Klasse *MetricCalculator*²²

private static MetricGenerator *generator*

Ein Objekt der singleton Klasse *MetricGenerator*²³

private static LinkedList<Metric> *allMetrics*

Die Liste aller verfügbaren Metriken

private static LinkedList<Result> *resultsViewInput*

Liste der Ergebnisse verschiedener Metrikenberechnungen

private static TableViewer *resultsViewer*

Ein Verweis auf das GUI Element in dem die Ergebnisse der Metrikenberechnungen angezeigt werden

Methoden:

public *EMFMetrics*()

Konstruktor mit dem eine neue Instanz von *EMFMetrics* erzeugt wird. Dieser wird in der Klasse *Activator* des Plugins aufgerufen.

²² Die Klasse *MetricCalculator* wird im Unterkapitel 6.2.3 genau beschrieben.

²³ Die Klasse *MetricGenerator* wird im Unterkapitel 6.2.5 genau beschrieben.


```
public static void calculateConfiguredMetrics(IProject project,  
                                             List<EObject> context)
```

Diese Methode berechnet alle für das angegebene Projekt spezifizierten Metriken die dem angegebenen Kontext entsprechen.

Um die Liste der zu berechnenden Metriken zu bekommen wird die *private* Methode *getConfiguration()* benutzt. Um die Berechnung durchzuführen wird die Instanz der Klasse *MetricCalculator* benutzt. Schließlich werden die Ergebnisse der Berechnungen abgefragt und zu dem *resultsViewInput* hinzugefügt. Dieser wird danach als neue Eingabe für den *resultsViewer* gesetzt..

Parameter:

project – das Projekt dessen Konfiguration von Metriken verwendet werden soll.

context – der Kontext auf dem die Berechnung stattfinden soll.

```
public static void createNewMetric(IProgressMonitor monitor, MetricInfo info)
```

Diese Methode erstellt eine neue Metrik. Dazu wird die Instanz der Klasse *MetricGenerator* benutzt.

Abschließend werden die Klassenabhängigkeiten des Zielprojektes aktualisiert und es wird ein Eintrag in die *plugin.xml*²⁴ Datei des Zielprojektes generiert. Dazu werden die Klassen *DependenciesFile*²⁵ und *XMLPluginFile* benutzt.

Diese Methode erwartet ein *monitor* als Parameter. Dieser wird dazu benötigt um den Fortschritt der Metrikengenerierung verfolgen zu können.

Parameter:

monitor – ein Monitor der den Generierungsfortschritt überwacht.

info – das Behälterobjekt mit den Daten der zu erstellenden Metrik.

```
public static Configuration loadConfiguration(IProject project)
```

Diese Methode lädt die Metrikenkonfiguration für das angegebene Projekt aus der *project.xml* Datei. Für den Ladevorgang wird die Klasse *XMLProjectFile* benutzt.

Parameter:

project – Das Projekt aus dessen *project.xml* Datei die Konfiguration geladen werden soll

24 Für eine genaue Beschreibung der XML-Dateien sehe Unterkapitel 6.4.3.

25 Für eine genaue Beschreibung der hier verwendeten Hilfsklassen sehe Unterkapitel 6.2.17.

```
public static void saveConfiguration(IProject project)
```

Diese Methode speichert die Metrikenkonfiguration für das angegebene Projekt in die `project.xml` Datei in dem Projektverzeichnis. Für den Speichervorgang wird die Utility-Klasse *XMLProjectFile* benutzt.

Parameter:

project – Das Projekt in dessen `project.xml` Datei die Konfiguration gespeichert werden soll

```
public static LinkedList<Metric> getAllMetrics()
```

Diese Methode liefert eine Liste aller verfügbaren Metriken. Sollten noch keine Metriken geladen werden, so wird dies gemacht.

```
public static LinkedList<Metric> getSelectedMetrics(IProject project)
```

Diese Methode liefert eine Liste der für das angegebene Projekt ausgewählten Metriken.

Es wird zunächst ein *Configuration*²⁶ Objekt für das angegebene Projekt gesucht und anschließend wird dessen Metrikenuswahl abgefragt.

Parameter:

project – das Projekt dessen Liste von Metriken geladen werden soll.

```
public static LinkedList<Metric> getFilteredMetrics(String metamodel,  
                                                    String context)
```

Diese Methode liefert die anhand von dem Metamodell und dem Kontext gefilterte Liste aller verfügbaren Metriken.

Parameter:

metamodel – das Metamodell auf das die Metriken beschränkt werden sollen.

context – der Kontext auf den die Metriken beschränkt werden sollen.

```
public static void setConfiguration(IProject project, boolean[] selection)
```

Mit dieser Methode wird eine projektspezifische Metrikenauswahl in das für das Projekt zuständige *Configuration* Objekt gespeichert.

Es wird zunächst ein *Configuration* Objekt für das angegebene Projekt gesucht und anschließend wird dessen Metrikenauswahl neu gesetzt.

Parameter:

project – Das Projekt in dessen *Configuration* Objekt die Konfiguration gespeichert werden soll.

selection – die Auswahl der Metriken in Form eines boolean Arrays.

²⁶ Die Klasse *Configuration* wird im Unterkapitel 6.2.7 genau beschrieben.

private static Configuration getConfiguration(IProject project)

Diese Methode durchsucht zunächst die liste der zwischengespeicherten Konfigurationen auf einen dem angegebenen Projekt entsprechenden Eintrag.

Sollte solch ein Eintrag in der zwischengespeicherten Liste nicht gefunden werden, so wird aus der project.xml Datei im Projektpfad des angegebenen Projektes geladen.

Sollte danach immer noch kein *Configuration* Objekt vorliegen (z.B. wurde keine project.xml Datei gefunden) so wird eine neuer *Configuration* Objekt erstellt.

Parameter:

project – Das Projekt für das eine Konfiguration gesucht wird.

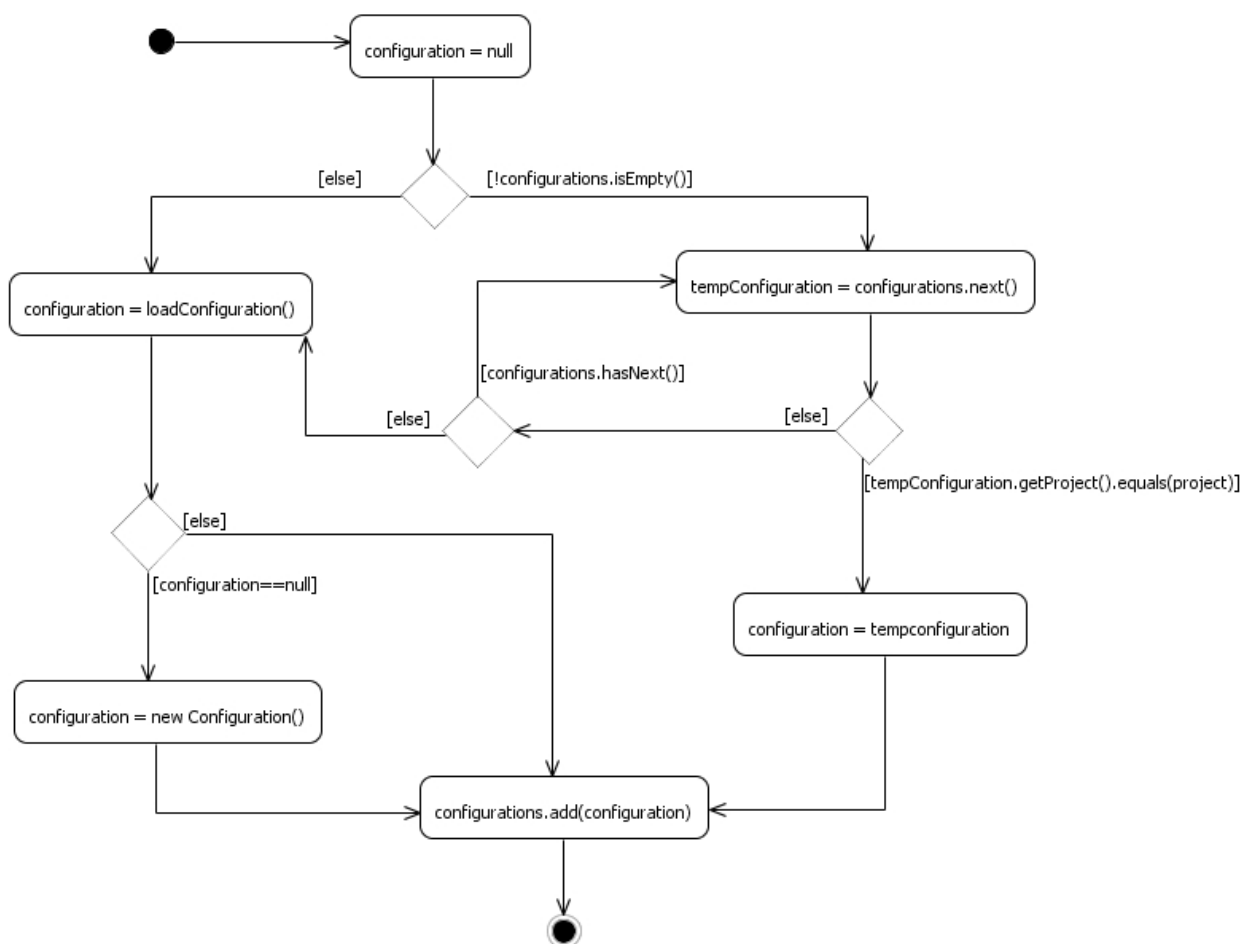


Abbildung 6.1. `getConfiguration()` Aktivitätsdiagramm

Bei den restlichen Methoden dieser Klasse handelt es sich um *getter* und *setter* Methoden für die Klassenattribute.

6.2.3. *MetricCalculator*

Diese Klasse bildet den für das Ausführen von Metrikberechnungen zuständigen Controller. Die Implementation dieser Klasse entspricht dem „Singleton“ Entwurfsmuster. So wird sichergestellt, dass zu jedem Zeitpunkt höchstens eine Instanz dieser Klasse existiert.

Erweiterungskompatibilität: Diese Klasse ist unabhängig von der Implementation der Metrikenberechnungsklasse, so können weitere Ansätze für die Berechnung von Metriken implementiert werden ohne diese Controller-Klasse verändern zu müssen.

Attribute:

private Configuration configuration

Das *singleton* Objekt dieser Klasse, dieses wird über die *getter* Methode an Klienten der Klasse weitergegeben.

private List<EObject> context

Das Kontext auf dem die Metrikenberechnung ausgeführt werden soll. Als Format wird hier eine Liste von *EObject* Elementen verwendet

private LinkedList<Result> results

Eine Liste von Ergebnissen. Die Ergebnisse werden als *Result*²⁷ Objekte gespeichert.

Methoden:

public static MetricCalculator getCalculator()

Mit dieser Methode kann das *singleton* Objekt angefordert werden. Sollte noch kein Objekt von der Klasse erstellt sein so wird der Konstruktor aufgerufen.

private MetricCalculator()

Der Konstruktor wird als private Methode implementiert um dem "Singleton" Entwurfsmuster gerecht zu werden. Dieser Konstruktor wird nur in der *getCalculator()* Methode aufgerufen.

public void calculateMetrics()

Mit dieser Methode werden alle in der *configuration* ausgewählten Metriken die dem *context* entsprechen berechnet. Um die einzelnen Metriken zu berechnen wird die private Methode *calculateMetric()* verwendet. Für die einzelnen Ergebnisse der Berechnung werden neue Objekte vom Typ *Result* erstellt und zu der *results* Liste hinzugefügt

²⁷ Die Klasse *Result* wird im Unterkapitel 6.2.8 genau beschrieben.

```
private Result calculateMetric(Metric metric)
```

Mit dieser Methode wird die angegebene Metrik berechnet. Für die Berechnung wird die zunächst die *ICalculateClass*²⁸ implementierende Klasse der Metrik verwendet.

Parameter:

metric – Die zu berechnende Metrik.

```
public void setConfiguration(Configuration configuration)
```

```
public void setContext(List<EObject> context)
```

Zwei *setter* Methoden für die Konfiguration den Kontext.

```
public LinkedList<Result> getResults()
```

Eine *getter* Methode für die Ergebnisliste.

6.2.4. *HenshinUtils*

Diese statische Klasse implementiert die Funktionalität die von den *ICalculateClass* implementierenden Klassen benötigt wird um Metrikenberechnungen anhand von Henshin-Transformationsregeln durchzuführen.

Erweiterungskompatibilität: Sollten weitere Ansätze für die Berechnung einer Metrik benötigt werden so können weitere den neuen Ansätzen entsprechende Klassen implementiert werden. Die Restlichen Komponente des Systems müssen nicht verändert werden.

Methoden:

```
public static double run(HenshinTransformationsPack pack, List<EObject> context)
```

In dieser Methode werden die im *HenshinTransformationsPack*²⁹ Objekt enthaltenen Transformationen auf dem angegebenen Kontext ausgeführt. Das Objekt besteht aus drei Transformationen. Diese werden auf dem Kontext Objekt der Reihe nach ausgeführt. Die erste und dritte Transformation ist optional. Die zweite liefert das Ergebnis der Metrikenberechnung.

Um die Transformationen auszuführen werden zwei *private* Methoden benutzt.

```
private static void process(String transf, List<EObject> context)
```

```
private static double count(String transf, List<EObject> context)
```

Die beiden in *run()* benutzten Methoden. Je nach dem Typ der Transformation³⁰ wird eine dieser Methoden verwendet um die Transformation auszuführen.

28 Das Interface *ICalculateClass* wird im Unterkapitel 6.2.13 genau beschrieben.

29 Die Klasse *HenshinTransformationsPack* wird im Unterkapitel 6.2.9 genau beschrieben.

30 Genauere Informationen zu den Transformationstypen befinden sich im Unterkapitel 6.4.2

6.2.5. *MetricGenerator*

Diese Klasse bildet den für das Erstellen einer neuen Metrik zuständigen Controller. Die Implementation dieser Klasse entspricht dem „Singleton“ Entwurfsmuster. So wird sichergestellt, dass zu jedem Zeitpunkt höchstens eine Instanz dieser Klasse existiert.

Diese Klasse verwendet Template-Dateien³¹ um die neuen Klassen mit den Code zur Metrikenberechnung zu generieren. Es werden zwei verschiedene Templates für die zwei verschiedenen Spezifikationsmöglichkeiten verwendet³². Alle generierten Klassen implementieren die *ICompositeCalculateClass*³³ Schnittstelle..

Erweiterungskompatibilität: Sollten weitere Spezifikationsmöglichkeiten gewünscht werden, so müssen nur weitere Templates hinzugefügt werden. Die interne Struktur dieser Klasse muss nicht verändert werden.

Attribute:

private static *MetricGenerator generator*

Das *singleton* Objekt dieser Klasse, dieses wird über die *getter* Methode an Klienten der Klasse weitergegeben.

private static *String templateDirectory*

Das Verzeichniss in dem sich die Templates für die zu erstellende Metrik befinden.

private static *List<IClasspathEntry> classpathEntries*

Eine Liste von Klassenpfad-Einträgen zu Klassen die für die Kompilierung der Template-Dateien gebraucht werden. Es handelt sich im *MetricInfo*³⁴ Klassen die Inhalte für die Templates bereitstellen.

private *MetricInfo metricInfo*

Der Behälter mit den Daten der zu erstellenden Metrik.

Methoden:

public static *MetricGenerator getGenerator()*

Mit dieser Methode kann das *singleton* Objekt angefordert werden. Sollte noch kein Objekt von der Klasse erstellt sein so wird der Konstruktor aufgerufen.

31 Die Template-Dateien werden im Unterkapitel 6.4.1 vorgestellt.

32 Die verschiedenen Spezifikationsmöglichkeiten wurden im Unterkapitel 5.3.6 vorgestellt.

33 Das Interface *ICompositeCalculateClass* wird im Unterkapitel 6.2.14 genau beschrieben.

34 Die Klasse *MetricInfo* wird im Unterkapitel 6.2.10 genau beschrieben.

```
private MetricGenerator()
```

Der Konstruktor wird als private Methode implementiert um dem "Singleton" Entwurfsmuster gerecht zu werden. Dieser Konstruktor wird nur in der *getCalculator()* Methode aufgerufen.

```
public void setMetricInfo(MetricInfo info)
```

Die setter Methode für den Behälter mit den Daten der zu erstellenden Metrik.

Parameter:

info - Der Behälter mit den Daten der zu erstellenden Metrik.

```
public void createCalculateClass(IProgressMonitor monitor)
```

Mit dieser Methode wird eine neue Klasse mit dem Berechnungscode für die angegebene Metrik erstellt. Diese Methode benutzt die beiden *private* Methoden: *generateCode()* und *saveCode()*.

Diese Methode erwartet ein *monitor* Objekt als Parameter. Es wird dazu benötigt um den Fortschritt der Metrikengenerierung verfolgen zu können.

Parameter:

monitor – ein Monitor der den Generierungsfortschritt überwacht.

Die beiden in *createCalculateClass* benutzten Methoden:

```
private String generateCode(IProgressMonitor monitor, String template)
```

Mit dieser Methode wird der Quellcode der neuen Klasse als String erstellt. Als Parameter erwartet sie den Namen des Templates das für die Metrik benutzt werden soll und einen *monitor*.

Parameter:

monitor – ein Monitor der den Generierungsfortschritt überwacht.

template – Name des Templates

```
private void saveCode(IProgressMonitor monitor, String content, String fileName)
```

Mit dieser Methode wird der durch *generateCode()* erstellte String schließlich in die entsprechende Datei gespeichert. Als Parameter erwartet sie den Inhalt und den Namen der Klasse die gespeichert werden soll sowie einen *monitor*.

Parameter:

monitor – ein Monitor der den Generierungsfortschritt überwacht.

content – Inhalt der neuen Klasse.

fileName – Name der neuen Klasse.

6.2.6. *Metric*

Diese Klasse entspricht der systeminternen Repräsentation einer Metrik nach dem „Composite“ Entwurfsmuster.

Um das Verändern der Metriken auszuschließen und gute Eigenschaften bezüglich der Datenkapselung zu gewährleisten, werden alle Klassenattribute als *private* implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden können.

Attribute:

private String id

Die einzigartige Identifikation der Metrik.

private String name

Der Name der Metrik.

private String description

Die Beschreibung der Metrik.

private String metamodel

Das Metamodell für das die Metrik angewendet werden kann.

private String context

Der Kontext in dem Die Metrik berechnet werden kann.

private String valueType

Der Typ des Wertes der als Ergebniss geliefert wird.

private ICalculateClass calculateClass

Ein Verweis auf die Klasse mit dem Code zur Berechnung der Metrik.

Methoden:

public Metric(String name, String description, String metamodel, String context, String valueType, ICalculateClass calculateClass, String id)

Der Konstruktork mit dem eine neue Instanz der Klasse erzeugt wird. Als Parameter werden die Werte für alle Klassenattribute erwartet.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

public String getId()

public String getName()

public String getDescription()

public String getMetamodel()

public String getContext()

public String getValueType()

public ICalculateClass getCalculateClass()

6.2.7. Configuration

Die Objekte dieser Klasse entsprechen der systeminternen Repräsentation von projektspezifischen Konfigurationen von Metriken. In dieser Klasse wird die Auswahl von Metriken für ein bestimmtes Projekt gespeichert.

Attribute:

private IProject project

Das Projekt zu dem die Konfiguration gehört.

private boolean[] selection

Die Auswahl an Metriken. Um Redundanz zu vermeiden und systeminterne Prozesse zu beschleunigen werden die Ausgewählten Metriken hier nicht direkt gespeichert. Es wird ein boolean Array mit Verweisen auf die Liste der zur Zeit verfügbaren Metriken im System.

Methoden:

public Configuration(IProject project)

Der Konstruktor mit dem ein neues *Configuration* Objekt erstellt wird. Dieser Konstruktor wird in der Klasse *EMFMetrics* aufgerufen. Als Parameter wird ein Verweis auf das Projekt zu dem die Konfiguration gehören soll erwartet.

public LinkedList<Metric> getSelectedMetrics()

Diese Methode liefert eine Liste der ausgewählten Metriken. Im Gegensatz zu dem Format des entsprechenden Attributes der Klasse wird hier eine Liste von *Metric* Objekten zurückgegeben. Diese wird anhand eines Vergleiches des Klasseninternen Arrays mit der durch *EMFMetrics* bereitgestellten Liste aller verfügbaren Metriken.

public void setSelection(**boolean[]** selection)

public void setSelection(LinkedList<String> idList)

Zwei setter Methoden zum setzen der Auswahl an Metriken. Im Falle der zweiten Methode wird als Parameter eine Liste von Metric Objekten erwartet, anhand von dieser Liste wird ein boolean Array erstellt. Dafür wird die private Methode *getSelection()* verwendet.

public IProject getProject()

getter Methode für das entsprechende Klassenattribut.

private boolean[] getSelection(LinkedList<String> idList)

Diese Methode liefert anhand von einer Liste von Metric Objekten ein Array mit Verweisen auf die Positionen der Metriken in der von *EMFMetrics* bereitgestellten Liste aller verfügbaren Metriken.

6.2.8. Result

Die Objekte dieser Klasse entsprechen der systeminternen Repräsentation von Ergebnissen einer Metrikenberechnung nach dem „Composite“ Entwurfsmuster. In den Objekten dieser Klasse werden alle vom System benötigten Informationen über das Ergebnis einer Metrikenberechnung gespeichert.

Um das Verändern der Ergebnisse auszuschließen und gute Eigenschaften bezüglich der Datenkapselung zu gewährleisten werden alle Klassenattribute als `private` implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden können.

Attribute:

private `Metric metric`

Die Metrik der das Ergebnis entspricht.

private `List<EObject> context`

Das Kontext auf dem die Metrik berechnet wurde.

private `String timeStamp`

Das genaue Datum an dem die Berechnung ausgeführt worden ist.

private double `resultValue`

Der Wert des Ergebnisses.

Methoden:

public `Result(Metric metric, List<EObject> context, double resultValue)`

Der Konstruktor mit dem ein neues *Result* Objekt erstellt wird. Dieser Konstruktor wird in der Klasse *MetricCalculator* aufgerufen. Als Parameter werden die Werte für die entsprechenden Klassenattribute erwartet. Zum setzen des genauen Datums wird die *CommonUtils*³⁵ Klasse verwendet.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

```
public String getTimeStamp()  
public Metric getMetric()  
public List<EObject> getContext()  
public double getResultValue()
```

35 Für eine genaue Beschreibung der hier verwendeten Hilfsklassen siehe Unterkapitel 6.2.17.

6.2.9. *HenshinTransformationsPack*

Um dem „Composite“ Entwurfsmuster gerecht zu werden, werden die von einer *ICalculateClass* implementierenden Klasse benutzten Transformationen in Pakete zusammengefasst. In den Objekten dieser Klasse werden Verweise auf jeweils drei Henshin-Transformationen gespeichert.

Um das Verändern der Transformationen auszuschließen und gute Eigenschaften bezüglich der Datenkapselung zu gewährleisten werden alle Klassenattribute als *private* implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden können.

Attribute:

```
private String prePath, tranfPath, postPath
```

Verweise auf die drei Transformationen.

Methoden:

```
public HenshinTransformationsPack(String prePath, String tranfPath,  
                                String postPath)
```

Ein Konstruktor mit dem ein neues Objekt der Klasse erstellt wird. Als Parameter werden die Werte für alle Klassenattribute erwartet.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

```
public String getPrePath()  
public String getTransformationPath()  
public String getPostPath()
```

6.2.10. *MetricInfo*

Die Objekte dieser Klasse bilden Behälter für die bei der Metrikenspezifikation eingegebenen Daten nach dem „Composite“ Entwurfsmuster.

Um das Verändern der Werte auszuschließen und gute Eigenschaften bezüglich der Datenkapselung zu gewährleisten werden alle Klassenattribute als *private* implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden können.

Dies ist eine abstrakte Klasse, die nur die Basisdaten jeder Metrik zusammenfasst. Für jede Art der Metrikenspezifikation wird eine Unterklasse von *MetricInfo* implementiert.

Erweiterungskompatibilität: Sollten weitere Möglichkeiten für eine Spezifikation einer neuen Metrik hinzugefügt werden so muss jeweils nur eine neue Unterklasse von *MetricInfo* implementiert werden.

Attribute:

private String project

Das Zielprojekt in das die Klasse mit dem Code zur Metrikenberechnung generiert werden soll

private String id

Die einzigartige Identifikation der zu erstellenden Metrik.

private String name

Der Name der zu erstellenden Metrik.

private String description

Die Beschreibung der zu erstellenden Metrik.

private String context

Der Kontext für zu erstellende Metrik.

private String metamodel

Das Metamodell der zu erstellenden Metrik.

private String valueType

Der Wertetyp der zu erstellenden Metrik.

Methoden:

```
public MetricInfo(String name, String id, String description, String metamodel,  
                  String context, String valueType, String project)
```

Ein Konstruktor mit dem ein neues Objekt der Klasse erstellt wird. Als Parameter werden die Werte für alle Klassenattribute erwartet.

```
public String getClassName()
```

Diese Methode liefert den Namen der zu erstellenden Metrik Klasse. Dieser besteht aus dem mit einer großer Buchstabe anfangendem Namen der Metrik.

```
public String getProjectName()
```

Diese Methode liefert den Namen des Projektes in das die neue Metrik-Klasse erstellt werden soll. Dieser ergibt sich aus dem Projektpfad.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

```
public String getPackage()  
public String getId()  
public String getName()  
public String getContext()  
public String getMetamodel()  
public String getValueType()  
public String getDescription()  
public String getProjectPath()
```

6.2.11. *HenshinMetricInfo*

Diese Klasse wird als eine Unterklasse von *MetricInfo* implementiert. Sie erweitert *MetricInfo* um die Eingabedaten die spezifisch für eine Metrikspezifikation durch die Angabe von Henshin Modelltransformationen.

Diese Klasse versorgt das *HenshinCalculateClassTemplate* mit Informationen.³⁶

Um das Verändern der Werte auszuschließen und gute Datenkapselungeigenschaften zu gewährleisten werden alle Klassenattribute als *private* implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden Können.

Attribute:

```
private String pre, transf, post;
```

Die Pfade zu den Henshin Modelltransformationen.

³⁶ Eine genaue Beschreibung der Templates befindet sich in dem Unterkapitel 6.4.1

Methoden:

```
public HenshinMetricInfo(String name, String id, String description,  
                        String metamodel, String context, String valueType,  
                        String project, String pre, String transf, String post)
```

Ein Konstruktor mit dem ein neues Objekt der Klasse erstellt wird. Als Parameter werden die Werte für alle Klassenattribute erwartet. Hier wird der Konstruktor der Oberklasse aufgerufen.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

```
public String getPreTransformationPath()  
public String getTransformationPath()  
public String getPostTransformationPath()
```

Die folgenden drei *getter* Methoden liefern die Namen der Transformationsdateien anhand von den gesamten Pfaden:

```
public String getPreTransformationName()  
public String getTransformationName()  
public String getPostTransformationName()
```

6.2.12. *CompositeMetricInfo*

Diese Klasse wird als eine Unterklasse von *MetricInfo* implementiert. Sie erweitert *MetricInfo* um die Eingabedaten die spezifisch für eine Metrikspezifikation aus vorhandenen Metriken.

Diese Klasse versorgt das *CompositeCalculateClassTemplate* mit Informationen.³⁷

Um das Verändern der Werte auszuschließen und gute Datenkapselungeigenschaften zu gewährleisten werden alle Klassenattribute als *private* implementiert. Es werden entsprechende *getter* Methoden bereitgestellt mit denen die Werte der Attribute abgefragt werden können.

Attribute:

```
private Metric firstMetric, secondMetric;
```

Die beiden Metriken die zusammengefügt werden.

```
private IOperation operation;
```

Die verbindende Operation.³⁸

³⁷ Eine genaue Beschreibung der Templates befindet sich in dem Unterkapitel 6.4.1.

³⁸ Für die metrikenverbindenden Operationen wurde eine Schnittstelle *IOperation* definiert. Eine genaue Beschreibung dieser Schnittstelle befindet sich im Unterkapitel 6.2.15.

Methoden:

```
public CompositeMetricInfo(String name, String id, String description, String  
                           metamodel, String context, String valueType,  
                           String project, Metric firstMetric,  
                           Metric secondMetric, IOperation operation)
```

Ein Konstruktor mit dem ein neues Objekt der Klasse erstellt wird. Als Parameter werden die Werte für alle Klassenattribute erwartet. In dem Konstruktor wird der Konstruktor der Oberklasse aufgerufen.

```
public String getCalculateOperationString()
```

Diese Methode liefert eine String Repräsentation des Rumpfes der *calculate()* Methode für das Template. Sie benutzt die von den *ICompositeCalculateClass* implementierenden Klassen der beiden Metriken bereitgestellte Methode *getCalculateOperation()* um die String Repräsentationen der jeweiligen *calculate()* Methoden zu bekommen. Anschließend setzt sie diese zusammen.

```
public String getTransformationsPacksString()
```

Diese Methode liefert eine String Repräsentation aller in den beiden Metriken verwendeten *HenshinTransformationsPack* Objekten. Sie wird von dem Template verwendet um den entsprechenden Inhalt zu generieren.

Diese Methode benutzt die private Methode *getTransformationsPacks()*

```
public String getOperationsString()
```

Diese Methode liefert eine String Repräsentation der *IOperation* implementierenden Objekte aus den beiden Metriken sowie des *IOperation* implementierenden Objektes das die beiden Metriken verbinden soll.

Sie wird von dem Template verwendet um den Inhalt des entsprechenden Attributes zu generieren.

Diese Methode benutzt die private Methoden: *getOperationString()* und *getOperationString()* um die String Repräsentationen der *IOperation* implementierenden Objekte zu bekommen.

```
private String getOperationString()
```

Diese Methode liefert eine String Repräsentation des *IOperation* implementierenden Objektes das Operation entspricht mit der die beiden Metriken verknüpft werden sollen.

```
private String getOperationsString(Metric metric)
```

Diese Methode liefert eine String Repräsentation aller in der angegebenen Metrik verwendeten *IOperation* implementierenden Objekten.

```
private String getTransformationsPacks(Metric metric)
```

Diese Methode liefert eine String Repräsentation aller in der angegebenen Metrik verwendeten *HenshinTransformationsPack* Objekten.

Die restlichen Methoden der Klasse sind *getter* Methoden für die Werte der Attribute:

```
public Metric getFirstMetric()  
public Metric getSecondMetric()  
public IOperation getOperation()
```

6.2.13. *ICalculateClass*

In dieser Schnittstelle werden die Anforderungen an Klassen mit dem Quellcode zur Metrikenberechnung spezifiziert. Diese Schnittstelle muss von allen Klassen die einer Metrik als Berechnungsklasse zugeordnet sind implementiert werden.

Diese Schnittstelle ermöglicht der Klasse *MetricCalculator* sich direkt auf die unten genannten Methoden der jeweiligen Klassen zu beziehen ohne die konkreten Implementationen der Objekte dieser Klassen zu kennen.

Methoden:

```
abstract public void setContext(List<EObject> context)
```

Die Methode mit der der Klasse ein Kontext übergeben werden kann.

```
abstract public double calculate()
```

Die Methode mit der die Metrikenberechnung ausgeführt werden kann.

6.2.14. *ICompositeCalculateClass*

Diese Schnittstelle wird als eine Unterklasse von *ICalculateClass* implementiert. Hier werden die zusätzlichen Anforderungen an die Klassen mit dem Quellcode zur Metrikenberechnung definiert, die die für das Zusammensetzen dieser Klassen erfüllt werden müssen.

Diese Schnittstelle ermöglicht die Definition neuer Metriken anhand zwei bereits bestehenden Metriken. Um keine Verweise auf die letzteren in der neuen Metrik zu haben, müssen gewisse Daten aus den „Vater-Metriken“ abgelesen werden. Diese Schnittstelle definiert die Methoden mit denen diese Daten abgefragt werden können.

Methoden:

abstract public String getCalculateOperation()

Die Methode mit der die String Repräsentation des Inhaltes der *calculate()* Methode der *ICalculateClass* implementierenden Klasse zurückgegeben wird.

abstract public HenshinTransformationsPack[] getTransformationPacks()

Die Methode mit der die String Repräsentation der Liste der Transformationspakete der *ICompositeCalculateClass* implementierenden Klasse zurückgegeben wird.

abstract public IOperation[] getOperations()

Die Methode mit der die String Repräsentation der Liste der Operationen der *ICompositeCalculateClass* implementierenden Klasse zurückgegeben wird.

6.2.15. IOperation

Um ein erweiterbares System an mathematischen Operationen für die Zusammensetzung zweier Metriken zu gewährleisten, wurde diese Schnittstelle definiert.

Jede mathematische Operation die als verknüpfende Operation bei der Zusammensetzung zweier Metriken funktionieren soll, wird als eine separate Klasse implementiert. Die Klassen werden mit Hilfe dieser Schnittstelle definiert.

Methoden:

public double calculate(double a, double b)

Die Methode die der mathematischen Operation der Klasse entspricht.

Im System wurden vier Klassen implementiert die diese Schnittstelle implementieren:

- *SumOperation*
- *SubtractionOperation*
- *MultiplicationOperation*
- *DivisionOperation*

Erweiterungskompatibilität: Sollte neben den vier oben genannten Operationen eine weitere mathematische Operation für die Zusammensetzung von Metriken benötigt werden, so muss nur eine neue Klasse die die *IOperation* Schnittstelle implementiert, implementiert werden

6.2.16 *MetricLoader*

Diese Klasse bildet den für das Laden von Metriken zuständigen Controller.

Diese Klasse wird als statische Klasse implementiert und bildet eine der Hilfsklassen.³⁹

Methoden:

```
public static LinkedList<Metric> loadMetrics()
```

Diese Methode ist für das Laden von Metriken über den Extension-Point⁴⁰ zuständig. Sie benutzt die private Methode *createMetric()*.

```
private static Metric createMetric(IConfigurationElement rawMetric)
```

Diese Methode erstellt ein neues *Metric* Objekt anhand von einem durch den Extension-Point gelieferten *IConfigurationElement* Objekt.

6.2.17. Hilfsklassen

Das System verfügt über zahlreiche Hilfsklassen die eine Palette von statischen Methoden implementieren. Diese werden hauptsächlich in den verschiedenen Controller-Klassen genutzt.

- ***CommonUtils*** – übliche durch verschiedene Klassen verwendete Methoden
- ***FileUtils*** – Methoden die sich auf das Dateisystem beziehen
- ***DependenciesFile*** – Methoden die dafür zuständig sind die Klassenabhängigkeiten eines Projektes zu verändern
- ***SelectionUtils*** – Methoden mit denen die aktuell ausgewählte Element im Editor oder Paketexplorer abgerufen werden kann
- ***ProjectUtils*** – Methoden die dafür zuständig sind das aktuelle Projekt zu liefern
- ***XMLFile*** – abstrakte Klasse die gemeinsame XML-Funktionen der XML-Klassen implementiert
- ***XMLResultFile*, *XMLProjectFile*, *XMLResultsFile*** – für das lesen und speichern der entsprechenden XML-Dateien zuständige Methoden

³⁹ Die anderen Hilfsklassen wurden im Unterkapitel 6.2.17 vorgestellt.

⁴⁰ Genauere Informationen zu dem Extension-Point Mechanismus befinden sich im Unterkapitel 6.4.4.

6.3. Sequenzdiagramme

Auf den folgenden Sequenzdiagramme werden die internen Systemabläufe aus der Sicht der einzelnen Objekte des Systems dargestellt. Jedes Diagramm entspricht einem der drei Hauptaktionen im System.⁴¹

Um die Diagramme übersichtlich zu gestalten wurde bei den Methodenaufrufen auf Parameter verzichtet. An einigen Stellen wurden mehrere Methoden zur einer zusammengefasst um den Zweck den sie erfüllen eindeutiger darzustellen. Die genauen Beschreibungen der tatsächlichen Methoden der jeweiligen Klassen wurden im Kapitel 6.2 vorgestellt.

6.3.1. Metriken konfigurieren

Der genaue Ablauf einer Metrikenkonfiguration wurde auf Abbildung 6.2 dargestellt.

Die Konfiguration einer projektspezifischen Metrikenauswahl wird durch den Benutzer über das „Properties“ Menü des entsprechenden Projektes in der GUI vorgenommen.

Das System verfügt über eine PropertyPage Klasse die für die Anzeige aller verfügbaren Metriken, sowie der Konfigurationen zuständig ist.

Bei dem anzeigen des „Properties“ Menü eines ausgewählten Projektes kommuniziert die Klasse *PropertyPage* zunächst mit *EMFMetrics* um an die Liste aller verfügbaren Metriken zu gelangen. Diese werden dann in der GUI angezeigt.

Um die projektspezifische Metrikenauswahl anzuzeigen muss die *PropertyPage* Klasse erneut eine Verbindung zu *EMFMetrics* aufbauen und die Liste aller Ausgewählten Metriken für das entsprechende Projekt abfragen. Um dieser Anfrage gerecht zu werden führt *EMFMetrics* zunächst die *getConfiguration()* Methode aus um an das *Configuration* Objekt für das angegebene Projekt zu gelangen. (siehe 6.2.2) Aus diesem Objekt kann *EMFMetrics* die projektspezifische Metrikenauswahl abfragen und an die *PropertyPage* klasse weitergeben. Diese übernimmt schließlich die Anzeige der Elemente auf der GUI.

Der Benutzer kann die Auswahl der Metriken verändern. Sobald er mit den Veränderungen fertig ist, wird die so entstandene Konfiguration gespeichert. Bei der Speicherung wird die neue Konfiguration an *EMFMetrics* gereicht. Dort wird die *getConfiguration()* Methode erneut aufgerufen aus um an das richtige *Configuration* Objekt für das aktuelle Projekt zu gelangen. In dieses Objekt kann *EMFMetric* letztendlich die neue Metrikenekonfiguration zwischenspeichern.

Wenn der Benutzer das „Properties“ Menu schließt, informiert *PropertyPage* *EMFMetrics* darüber,

⁴¹ Es werden die folgenden Aktionen gemeint, die den Anwendungsfällen entsprechen: Metriken konfigurieren, Metriken berechnen, Metrik spezifizieren.

dass das *Configuration* Objekt auf die Festplatte gespeichert werden soll. Letzteres geschieht mit Hilfe der *project.xml* Datei.

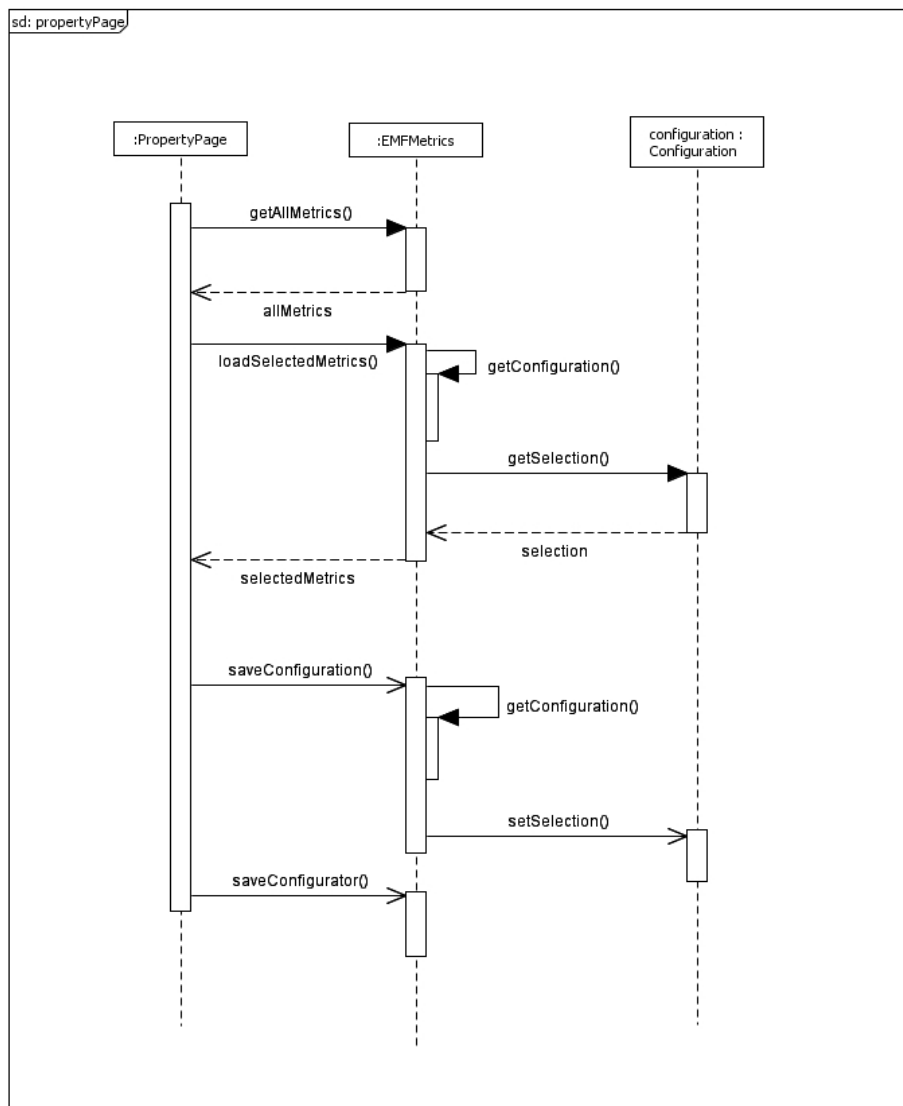


Abbildung 6.2. Metriken konfigurieren Sequenzdiagramm

6.3.2. Metriken berechnen

Der genaue Ablauf einer Metrikenberechnung wurde auf Abbildung 6.3 dargestellt.

Die Berechnung der Metriken wird von dem Benutzer initialisiert, indem er über ein Kontextmenü im baumbasierten Modelleditor die Aktion „Calculate metrics“ startet. Das Kontextmenü wird über einem bestimmten Element des zu untersuchenden Softwaremodells erstellt.

Die Klasse *CalculateMetricsActionDelegate* ist dafür zuständig dieses Element, sowie das Projekt in dem es sich befindet abzufangen und an den Hauptcontroller *EMFMetrics* weiterzuleiten. Dies geschieht bei dem Aufruf der Methode *calculateConfiguredMetrics()*. *EMFMetrics* ist für die Ausführung der kompletten Berechnung aller zutreffenden Metriken verantwortlich.

Als erstes führt *EMFMetrics* die Methode *getConfiguration()*⁴² mit dem Projekt als Parameter aus. Diese liefert ein Objekt der Klasse *Configuration*, im dem eine projektspezifische Metrikenauswahl gespeichert ist.

Dieses Objekt wird an die Klasse *MetricCalculator* weitergereicht. Des Weiteren wird *MetricCalculator* mit dem abgefangenem Element des Softwaremodells auf dem die Berechnung ausgeführt werden soll versorgt. Ein so initialisierter *MetricCalculator* kann die eigentliche Metrikenberechnung durchführen. Dies geschieht mit dem Aufruf der Methode *calculateMetrics()*.

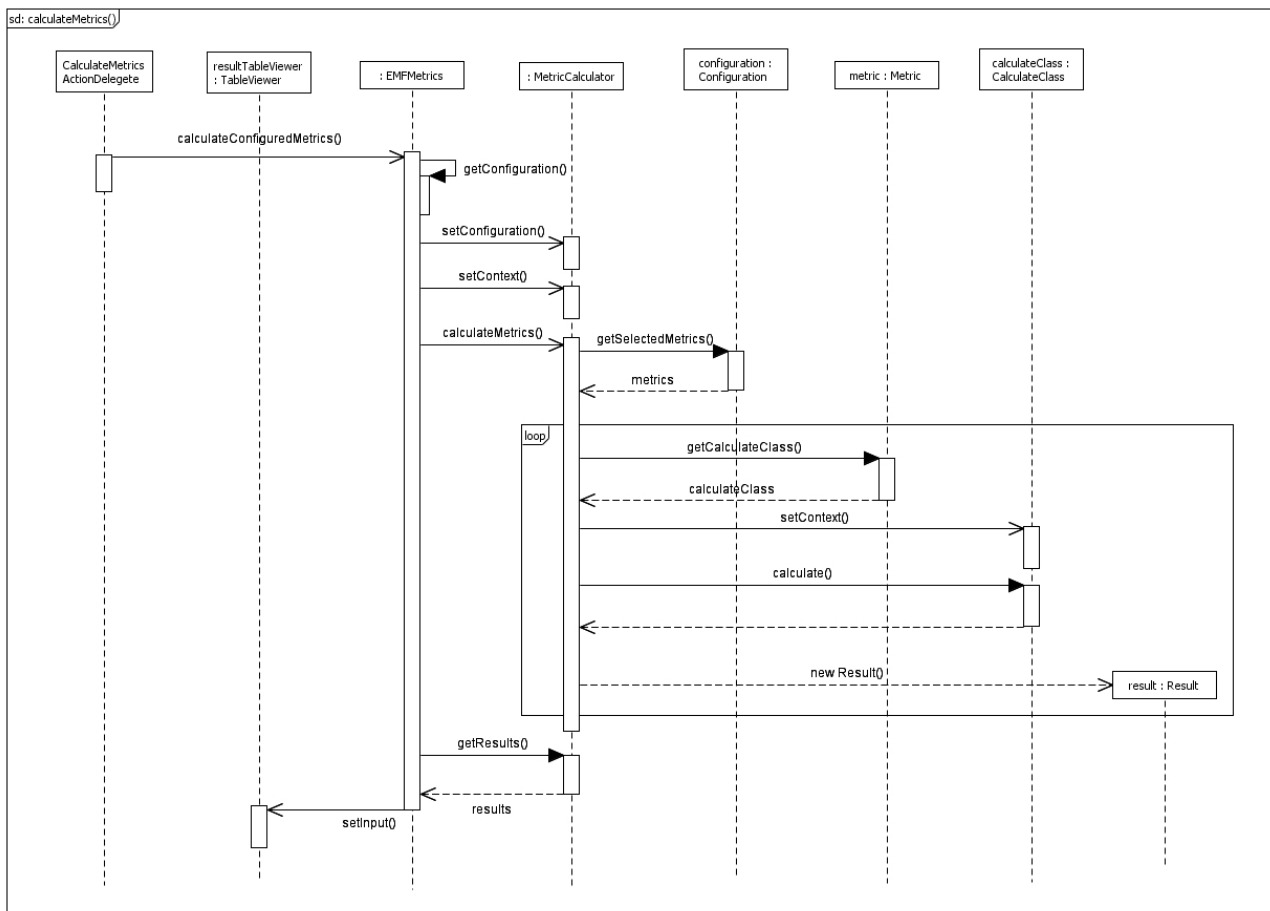


Abbildung 6.3. Metrik berechnen Sequenzdiagramm

Die Klasse *MetricCalculator* holt aus dem *Configuration* Objekt die zu berechnende Metrikenauswahl und überprüft jede dort enthaltene Metrik auf Kompatibilität mit dem Element des Softwaremodells auf dem die Berechnung ausgeführt werden soll.

Die daraus entstandene Liste von Metriken wird der Reihe nach abgearbeitet und für jede Metrik eine Berechnung ausgeführt.

⁴² Der genaue Ablauf der Methode *getConfiguration()* wurde im Unterkapitel 6.2.2 geschildert.

Um eine einzelne Metrikenberechnung auszuführen wird zunächst das jeweilige *Metric* Objekt nach der *CalculateClass* in der sich der Code zur Metrikenberechnung befindet abgefragt.

Das Objekt von *CalculateClass* wird dann mit dem Element des Softwaremodells initialisiert um schließlich die Berechnung darauf durchzuführen. Für das Ergebnis der Berechnung wird ein neues Objekt der Klasse *Result* erstellt.

Nachdem die Liste der zu berechnenden Metriken komplett durchgearbeitet worden ist, werden alle *Result* Objekte durch *EMFMetrics* abgefragt. Anhand von dieser Liste wird das *resultTableViewer* Objekt aktualisiert. Dieses ist für die Anzeige der Ergebnisse auf der GUI zuständig.

6.3.3. Metrik spezifizieren

Der genaue Ablauf einer Metrikenkonfiguration wurde auf Abbildung 6.1 dargestellt.

Die Spezifikation einer neuen Metrik kann von dem Benutzer auf zwei verschiedenen Wegen initialisiert werden. Zum einen indem er über das „File“ Menü die Aktion „New“->„Other“->„EMF Metrics“->„Metric“ startet.

Zum anderen indem er über ein Kontextmenü im baumbasierten Modelleditor die Aktion „Specify metrics“ startet. Das Kontextmenü wird über einem bestimmten Element des zu untersuchenden Softwaremodells erstellt.

Sollte der zweite Weg gewählt worden sein, so wird das Formular mit Basisdaten dem Element entsprechend gefüllt. Metamodell und Kontext werden gesetzt.

Die Klasse *CreateMetricActionDelegate* ist dafür zuständig einen neuen Wizard zu erstellen und diesem die benötigten Informationen zu übergeben.

Es wird ein Objekt der Klasse *NewMetricWizard* mit dem Namen *wizard* erstellt und initialisiert. Dieser erstellt zunächst jeweils ein Objekt von einer der drei Unterklassen von *WizardPage* *MetricDataWizardPage*, *HenshinDataWizardPage* und *CompositeDataWizardPage*.

Die Klasse *MetricDataWizardPage* stellt die erste Seite des Wizards bereit, die ein Formular für die Eingabe der Basisdaten der zu erstellenden Metrik enthält. Das Formular wird zunächst mit einigen Inhalten gefüllt. Zuerst werden alle aktiven Projekte in das entsprechende Feld hinzugefügt. Danach wird das Metamodell und der Kontext gefüllt. Die werte der drei Parameter werden vom *wizard* geholt.

HenshinDataWizardPage erstellt bei der Initialisierung eine Liste von verfügbaren Henshin Transformationsdateien. Diese wird anhand des aktuellen Projektes erstellt. Das aktuelle Projekt wird vom *wizard* geholt

CompositeDataWizardPage erstellt bei der Initialisierung eine Liste von verfügbaren Metriken anhand von dem über die *dataPage* angegebenen Metamodell und Kontext. Das Metamodell und

der Kontext werden vom *wizard* geholt. Die Liste der Metriken wird über den Hauptcontroller *EMFMetrics* geholt. Dieser filtert alle verfügbaren Metriken bezüglich des Metamodells und des Kontextes und gibt diese an die *compositePage* weiter.

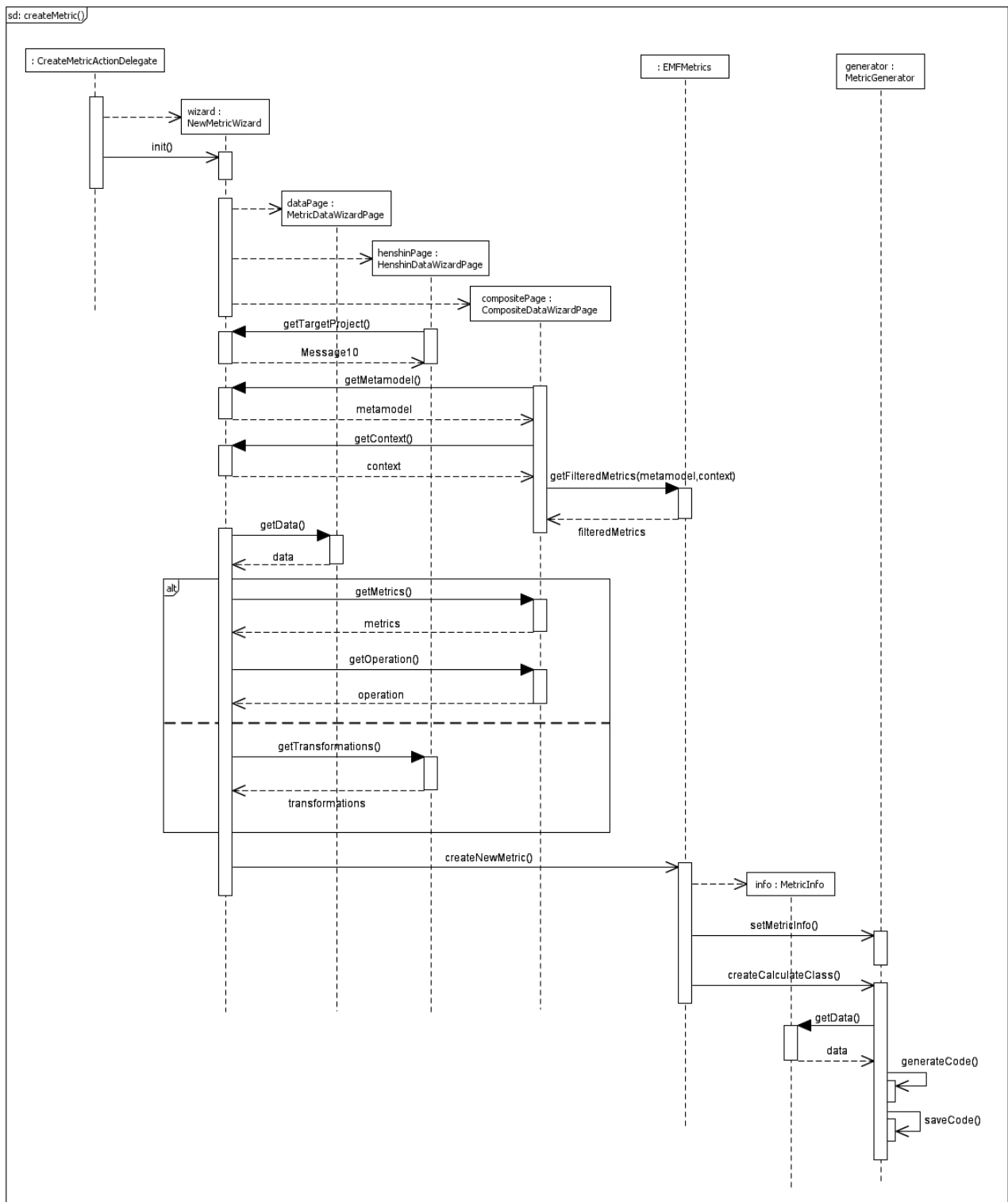


Abbildung 6.4. Metrik spezifizieren Sequenzdiagramm

Die Klassen *HenshinDataWizardPage* und *CompositeDataWizardPage* stellen die zweite Seite des Wizard bereit. Je nachdem für welche Art der Spezifikation sich der Benutzer entscheidet wird das entsprechende Fenster eingeblendet.

Nachdem der Benutzer alle erforderlichen Daten in die entsprechenden Seiten des Wizards eingegeben hat, wird die neue Metrik erstellt.

Bei der Erstellung werden zunächst alle Daten von dem *dataPage* Objekt abgefragt, danach wird die zweite Seite des Wizards abgefragt. Je nachdem welche Art der Spezifikation der Benutzer gewählt hat ist es das *henshinPage* oder das *compositePage* Objekt.

Im Falle einer Spezifikation durch die Angabe von einer Transformationsdatei wird *henshinPage* nach den Transformationen abgefragt. Sollte der Benutzer eine Spezifikation aus vorhandenen Metriken ausgewählt haben, so wird *compositePage* nach den beiden Metriken und der Kompositionsoption abgefragt.

Mit diesen gesammelten Daten als Parameterliste wird schließlich die Methode *createNewMetric()* in *EMFMetrics* ausgeführt. Diese erstellt zunächst ein Objekt der Klasse *MetricInfo* erstellt. Das *MetricInfo* Objekt wird an die Klasse *MetricGenerator* weitergegeben.

MetricGenerator ist für die Generierung der Klasse mit dem Code zur Metrikenberechnung zuständig. Anhand von den Daten aus *MetricInfo* wird der Inhalt der Klasse generiert und schließlich gespeichert.

6.4. Weitere Komponenten

Neben der Implementationen der einzelnen Klassen besteht das System aus zahlreichen anderen Elementen. Dabei handelt es sich um externe Schnittstellen oder pluginspezifische Dateien. Im folgenden werden die wichtigsten dieser Elemente vorgestellt.

6.4.1 „javajet“ Template Dateien

Die Template-Daten bilden Muster für die von der Klasse *MetricGenerator* zu erstellenden Klassen. Diese implementieren die *ICompositeCalculataClass* Schnittstelle um eine spätere Zusammensetzung der dazugehörenden Metriken zu ermöglichen.

In der aktuellen Implementierung werden zwei Templates benutzt. Ein für eine herkömmliche Klasse zur Berechnung von Metriken anhand von Henshin Transformationen und ein für die aus einer Zusammensetzung zweier Metriken entstandene Klasse.

Die Templates werden mit entsprechenden *MetricInfo* Objekten versorgt aus denen sie bei der Generierung alle benötigten Daten abfragen.

6.4.2 „henshin“ Transformationdateien

Die anhand von den Templates erstellten Klassen beziehen sich auf Henshin Transformationen um die Metrikenberechnungen durchzuführen. In gewissen Fällen ist es nicht möglich eine bestimmte Metrik mit nur einer Transformationsregel zu berechnen. Deswegen werden zwei Sorten von Transformationen verwendet:

- Transformationen die eine Metrikenberechnung durchführen
- Transformationen die das Modell auf dem eine Metrikenberechnung durchgeführt werden soll für die Berechnung vorbereiten

Sollte eine Metrik nicht durch eine einzelne Transformation zu berechnen sein, so wird ein entsprechendes Paket an Transformationen bereitgestellt. Die Interne Systemstruktur ist für aus drei Transformationen bestehende Pakete vorbereitet. So kann eine vorbereitende, eine berechnende und eine die Vorbereitungen wieder zurücknehmende Transformation für jede Metrik definiert werden.

Um Kompatibilität zu gewährleisten müssen von den Dateien mit den Transformationen die folgenden Eigenschaften erfüllt werden:

- In Berechnungstransformationen muss eine Regel mit dem Namen „mainRule“ definiert sein.
- In Vorbereitungstransformationen muss eine *TransformationUnit* mit dem Namen „mainUnit“ definiert sein.

- Es muss immer ein Parameter mit dem Namen „selectedEObject“ definiert sein, dieser soll einen Verweis auf das, dem Kontext entsprechende Objekt in der Transformationsregel liefern.

6.4.3. XML-Dateien

Vom System werden mehrere Dateien in dem XML-Format verwendet.

- Eine Datei „plugin.xml“ in der alle Informationen über von dem EMF Metrics Plugin bedienten Extensions gespeichert werden.
- Die Dateien „plugin.xml“ von Projekten in denen Metriken spezifiziert werden. Über diese Dateien werden die Metriken in der *MetricLoader* Klasse geladen.
- Eine Sammlung von Dateien „project.xml“ in denen die projektspezifische Metrikenauswahl der jeweiligen Projekte gespeichert wird.
- Eine weitere Sammlung von Dateien „results.xml“ in denen die Ergebnisse der Metrikenberechnungen gespeichert werden können.

6.4.4 „metric.exsd“ Datei

In dieser Datei wird das Schema der „Metric“-Extension definiert. Alle Projekte in denen Metriken spezifiziert werden, müssen diese Extension durch einen entsprechenden Extension-Point bedienen.

Die in „metric.exsd“ definierte Extension soll die folgenden Elemente beinhalten:

- metric_id - eine eindeutige Identifikation der Metrik
- metric_name - der Name der Metrik
- metric_description - die Beschreibung der Metrik
- metric_metamodel - das Metamodell für das die Metrik berechnet werden soll
- metric_context - der Kontext in dem die Metrik berechnet werden soll
- metric_value_type - der Typ des Wertes den die Metrik als Ergebnis liefern wird
- metric_calculate_class - ein Verweis auf die ICalculateClass implementierende Klasse die für die Berechnung der Metrik zuständig ist

Bei der Erstellung einer neuen Metrik wird ein Eintrag in die Datei „plugin.xml“ des entsprechenden Projektes mit den Werten dieser Attribute generiert.

7. Benutzerhandbuch

In diesem Kapitel wird eine Anleitung für die Benutzer von „EMF Metrics“ präsentiert.

Es werden zunächst alle in den Anforderungen (Kap. 4) spezifizierten Anwendungsfälle anhand der aktuellen Version des Systems erklärt. Anschließend werden anhand von konkreten Beispielen, die wichtigsten Funktionen vorgestellt.

Dieses Dokument enthält Antworten zu den meisten Fragen, die bei der Benutzung von EMF Metrics auftreten können.

7.1 Metriken konfigurieren

Um eine projektspezifische Konfiguration von Metriken zu erstellen muss zunächst die Property-Page des Projektes geöffnet werden. Diese kann über das Project-Menü oder über das Kontextmenü des Projektes erreicht werden (Abb. 7.1).

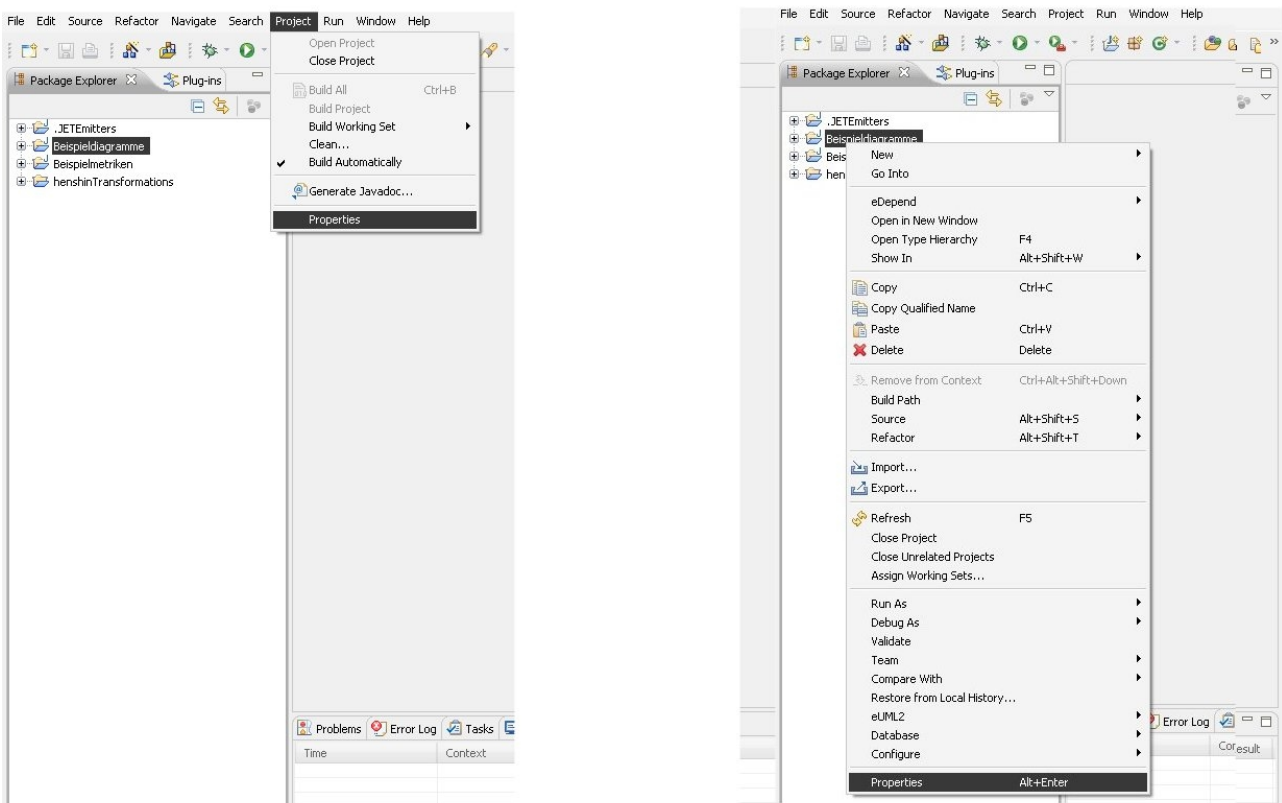


Abbildung 7.1. Screenshot: Properties.

Anschließend kann aus der Liste auf der linken Seite „EMF Metrics“ ausgewählt werden. Dort werden alle bei der Systeminitialisierung geladenen Metriken angezeigt und nach Metamodell in Tabs gruppiert (Abb. 7.2). Jeder Metrik ist eine checkbox zugeordnet über die, die jeweilige Metrik

ausgewählt werden kann. In der Tabelle werden die folgenden Attribute der Metriken angezeigt :

- Name der Metrik,
- Beschreibung der Metrik,
- Kontext in dem die Metrik berechnet werden kann.,
- Das Metamodell wird als Name des jeweiligen Tabs angezeigt.

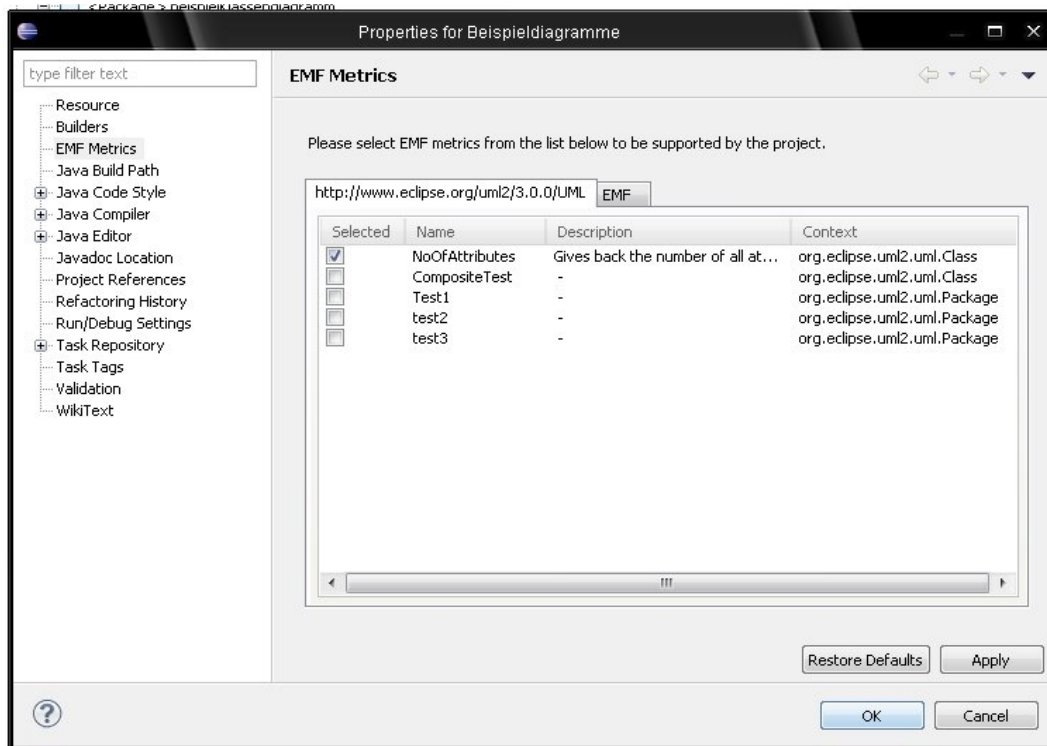


Abbildung 7.2. Screenshot: Properties-Menü.

Hier wird für das Projekt „Beispieldiagramme“ die Metrik „NoOfAttributes“ ausgewählt. Diese Metrik berechnet die Anzahl der Attribute einer Klasse und muss somit im Kontext einer Klasse angewendet werden. Als Metamodell wird hier UML benutzt.

Die so entstandene Auswahl wird nach dem klicken des „OK“ Buttons gespeichert und bleibt auch für weitere Sitzungen erhalten. Sie muss nicht für jede Sitzung neu erstellt werden, sie kann aber jeder Zeit auf die hier geschilderte Weise verändert werden.

Als default-Konfiguration für neue Projekte werden alle Metrik deaktiviert.

7.2. Metriken berechnen

Nachdem eine projektspezifische Metrikenkonfiguration erstellt worden ist können die dort ausgewählten Metriken für ein konkretes Modellelement (das dem Metamodel und dem Kontext entspricht) berechnet werden. Sollte eine Konfiguration in einer früheren Sitzung erstellt worden sein so wird diese automatisch geladen (es ist nicht notwendig wieder das Property-Menü zu öffnen).

7.2.1. Berechnung starten

Die Metrikenberechnung wird über das Kontextmenü des baumbasierten Modelleditor gestartet. Dazu muss dort zunächst das Untermenü „EMF Metrics“ und anschließend „Calculate metrics“ ausgewählt werden (Abb. 7.3). Dieses Untermenü wird nur für *EObject* Elemente angezeigt, um sicherzustellen, dass Metrikenberechnungen auf Objekten auf den sich auch berechnet werden können, gestartet werden.

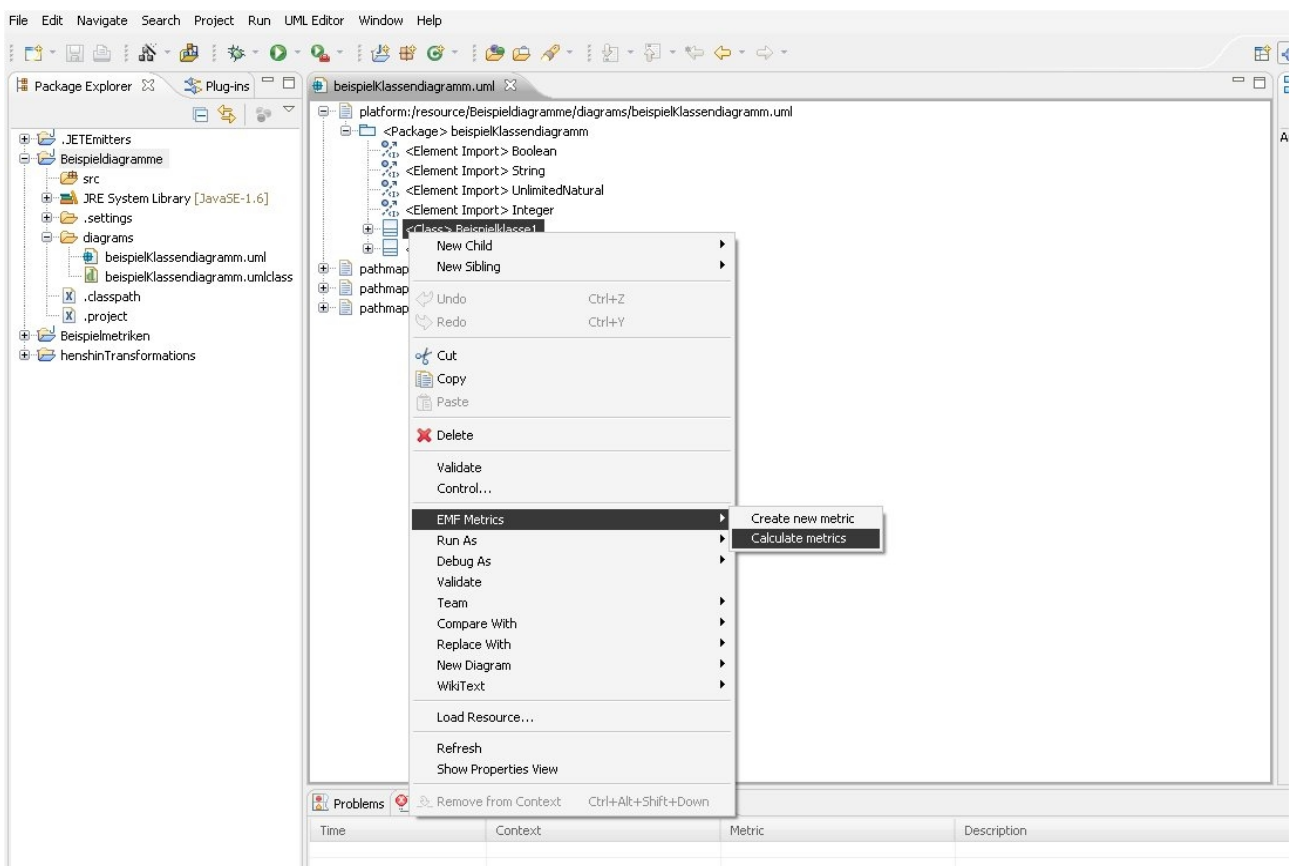


Abbildung 7.3. Screenshot: Kontextmenü vom baumbasierten UML-Editor:

Hier wurde ein Klassendiagramm mit dem Namen „Beispielklassendiagramm“ aus dem Projekt „Beispieldiagramme“ im baumbasierten Editor geöffnet, danach wurde eine konkrete Klasse des

Diagramms ausgewählt und durch das Kontextmenü dieses Elementes wird die Metrikenberechnung gestartet.

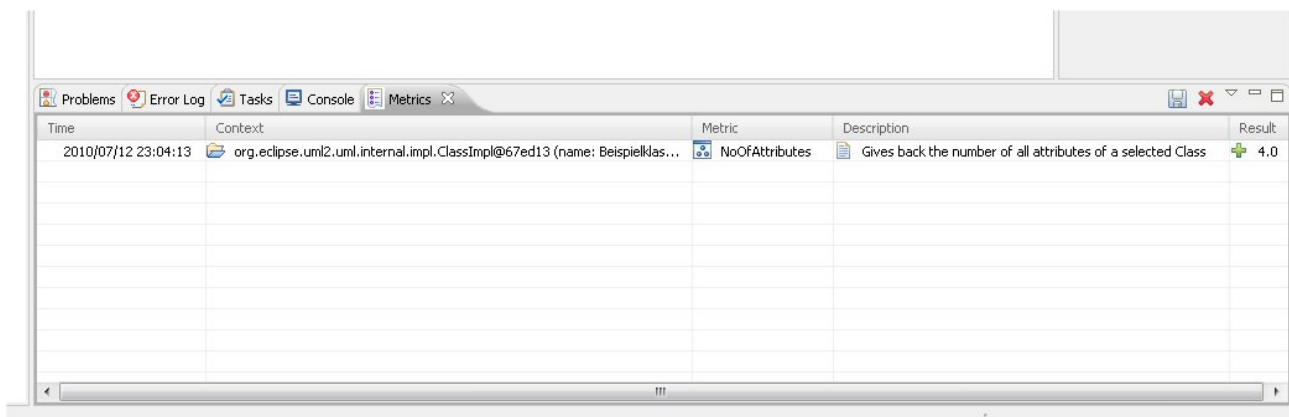
Nach dem starten der Berechnung werden zunächst alle für das entsprechende Projekt konfigurierten Metriken auf deren Kontext und Metamodell untersucht, und anschließend werden die, dessen Kontext und Metamodell dem, des ausgewählten Elementes entspricht berechnet.

7.2.2. Anzeige der Berechnungsergebnisse

Nach der Ausführung der Metrikenberechnungen werden die entsprechenden Ergebnisse in einer speziell dazu vorgesehenem Metrics-View angezeigt (Abb.7.4).

Für jede Metrik die berechnet wurde wird ein neuer Eintrag zu der sich dort befindenden Liste hinzugefügt. Die Einträge bestehen aus den folgenden Feldern:

- genaues Datum der Berechnung
- eine Repräsentation des Elementes auf dem die Berechnung ausgeführt worden ist
- Name der Metrik die berechnet wurde
- Beschreibung der Metrik die berechnet wurde
- Ergebnis der Berechnung



The screenshot shows the Eclipse IDE's Metrics-View. The view has a tab bar at the top with 'Metrics' selected. Below the tab bar is a table with the following columns: Time, Context, Metric, Description, and Result. The first row contains the following data:

Time	Context	Metric	Description	Result
2010/07/12 23:04:13	org.eclipse.uml2.uml.internal.impl.ClassImpl@67ed13 (name: Beispielklas...)	NoOfAttributes	Gives back the number of all attributes of a selected Class	4.0

Abbildung 7.4. Screenshot: Metrics-View.

In diesem Fall wurde die Metrik mit dem Namen „NoOfAttributes“ auf der Klasse mit dem Namen „Beispielklasse“ berechnet. Die Klasse besitzt 4 Attribute.

Die Ergebnisse weiterer Berechnungen werden zu den schon in der Liste vorhandenen hinzugefügt.

7.2.3. Speichern der Ergebnisse aus der Liste

Nachdem alle gewünschten Berechnungen ausgeführt worden sind, kann die Liste der Ergebnisse gespeichert werden. Dazu kann entweder die entsprechende Ikone geklickt werden oder das Kontextmenü des Entsprechenden Eintrages (Abb. 7.5).

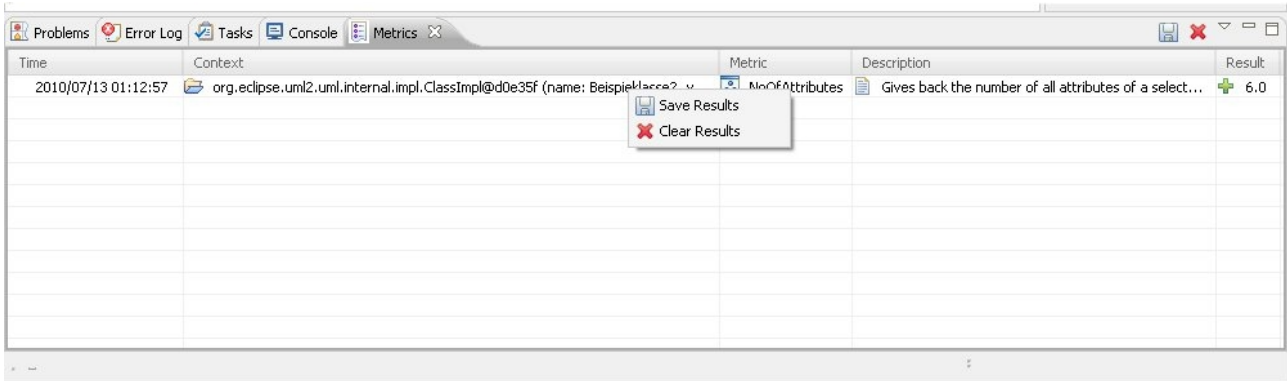


Abbildung 7.5. Screenshot: Metrics-View Kontextmenü.

Zur Speicherung der Liste der Ergebnisse wird ein Dateidialogfenster angezeigt in den der Benutzer gebeten wird einen Dateinamen und den Speicherort für die neue Datei anzugeben (Abb.7.6).

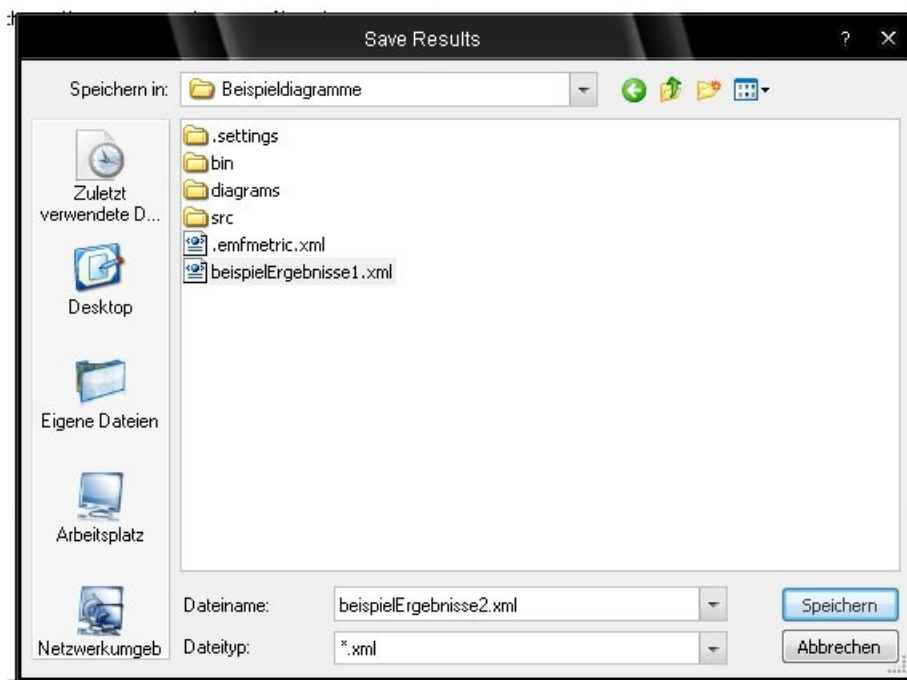


Abbildung 7.6. Screenshot: „Save results“ Dialogfenster.

Hier werden die Ergebnisse in eine Datei „beispielErgebnisse2.xml“ in das Verzeichnis des „Beispieldiagramme“ Projektes gespeichert .

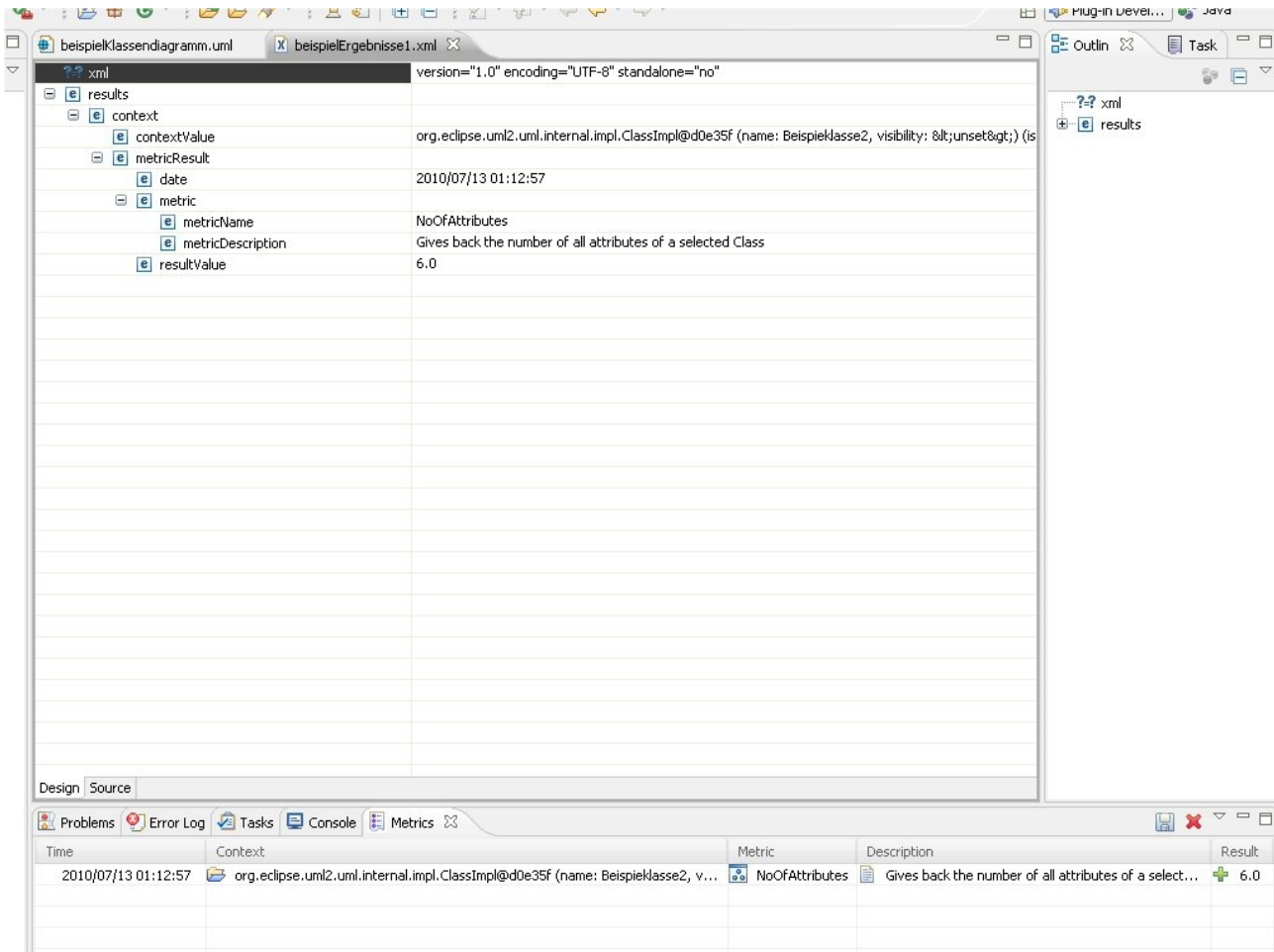


Abbildung 7.7. Screenshot: Inhalt der gespeicherten Datei.

Auf Abbildung 7.7 wurde der Inhalt der so entstandenen Datei in einem XML-Editor angezeigt. Als Vergleich unten in der Metrics-View die Liste der gespeicherten Metriken.

In diesem Fall wurde nur eine Metrik gespeichert.

7.2.4. Löschen der Ergebnisse aus der Liste

Die Liste der Ergebnisse kann analog zur Speicherung auch gelöscht werden. Dazu können die auf Abbildung 7.5. gezeigte Ikone oder alternativ das Kontextmenü der Metrics-View genutzt werden.

7.3. Metriken spezifizieren

Eine neue Metrikspezifikation kann auf zwei verschiedenen Wegen gestartet werden:

- Zum einen kann man über das File-Menü in zunächst das Untermenü „New“ und anschließend „Other...“ auswählen. (Abb.7.8). Danach wird ein Dialogfenster angezeigt in dem unter „EMF Metrics“ die Option „Metric“ ausgewählt werden kann (Abb.7.9).

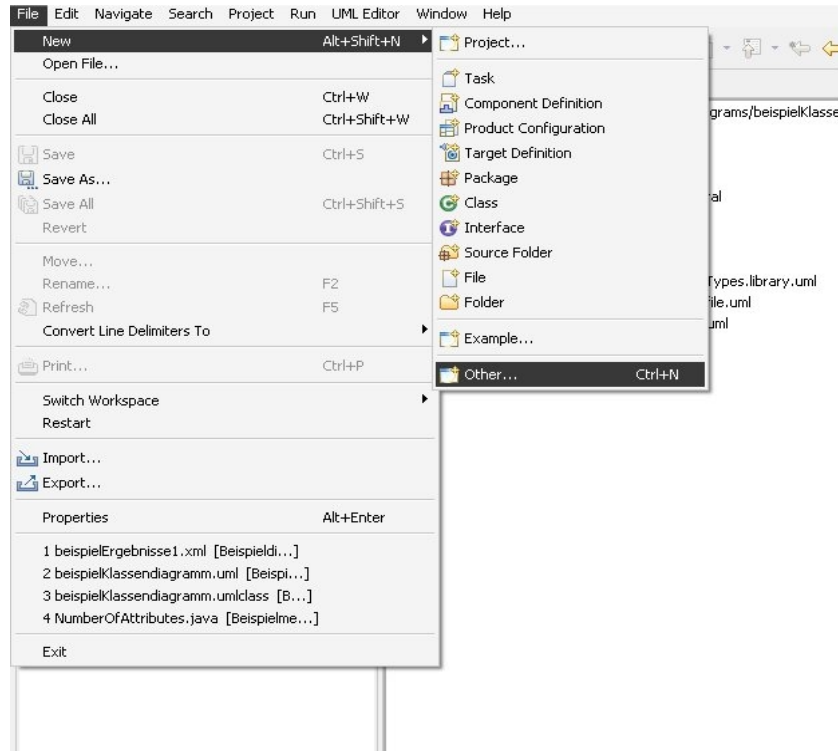


Abbildung 7.8. Screenshot: File-Menü.

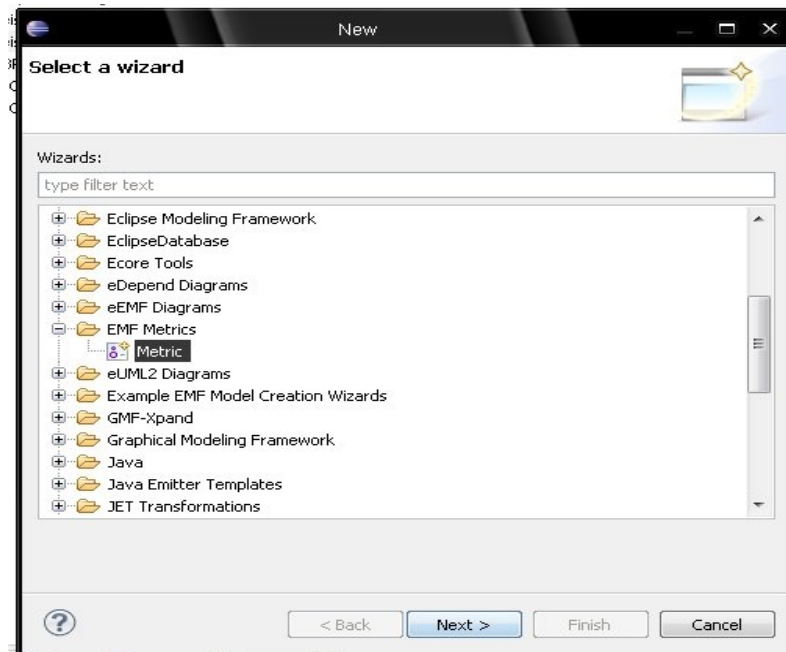


Abbildung 7.9. Screenshot: New-Dialogfenster.

- Zum anderen kann man direkt aus einem baumbasierten Modelleditor über das Kontextmenü zunächst das Untermenü „EMF Metrics“ und anschließend „Create new metric“ auswählen (Abb. 7.10)

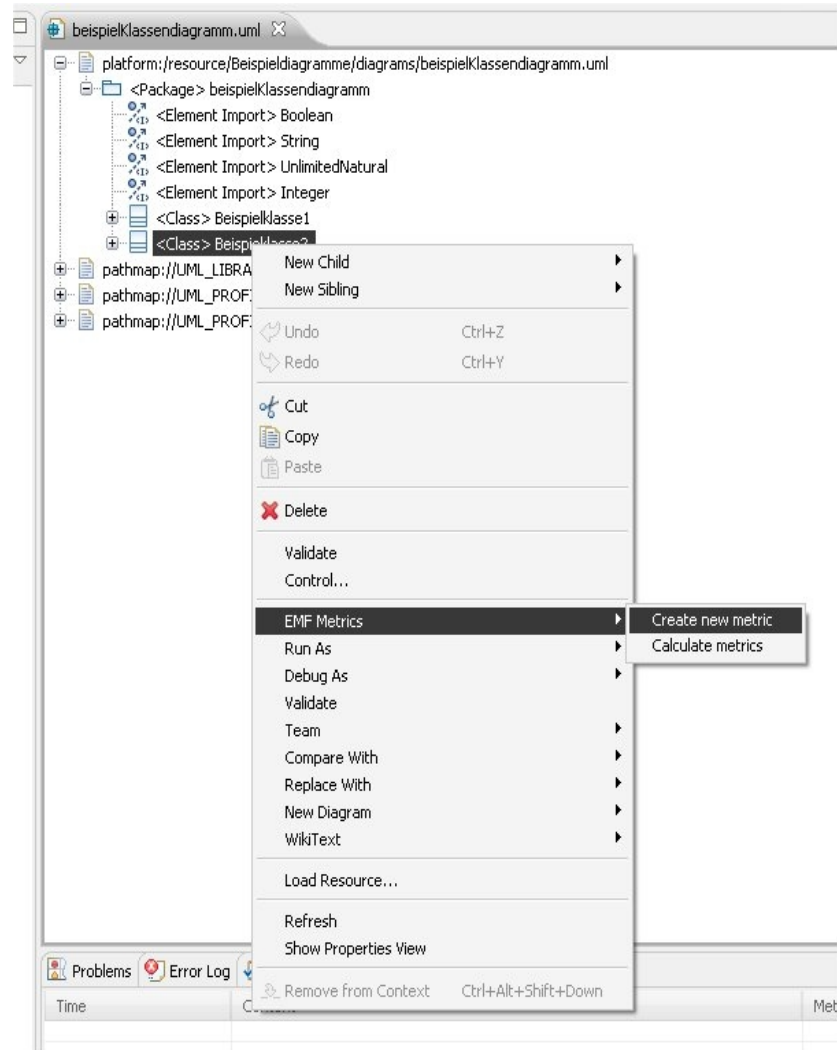


Abbildung 7.10. Screenshot: Kontextmenü vom baumbasierten UML-Editor.

7.3.1. Metric Data

Als nächstes wird ein Wizard eingeblendet. Die erste Seite des Wizards beinhaltet ein Formular in dem die Basisdaten der zu erstellenden Metrik angegeben werden sollen (Abb. 7.11).

Sollte die zweite Option aus dem vorherigen Unterkapitel genutzt worden sein, so wird das Dialogfenster mit dem Metamodell und Kontext des Elementes, über dem das Kontextmenü aufgemacht worden ist, gesetzt. Andernfalls sind alle Felder des Formulars leer.

Tipp: Es wird empfohlen die zweite Option aus dem vorherigen Unterkapitel zu nutzen um den Wizard zu starten, um die Angaben vom Metamodell und Kontext im richtigen Format zu bekommen.

Abbildung 7.11. Screenshot: New-Metric Wizard: Metric Data.

Das Formular soll mit den folgenden Daten gefüllt werden:

- einem Verweis auf das Zielprojekt in das die neue Metrik generiert werden soll. Hierfür wird eine Dropdown-Box verwendet. Diese wird mit allen zur Zeit verfügbaren Plugin Projekten gefüllt,
Achtung: Metriken können nur in Plugin Projekte generiert werden.
- dem Namen der neuen Metrik,
- einer Beschreibung der neuen Metrik,
- dem Metamodell für das die neue Metrik später berechnet werden soll,
- dem Kontext in dem die neue Metrik später berechnet werden soll,
- dem Wertetyp der Ergebnisse der Metrik (dieses Feld muss nicht gefüllt werden, als default-Wert wird hier „double“ verwendet),
- der Art auf die, die neue Metrik spezifiziert wird. (durch Angabe von Henshin Transformationsdateien oder durch Zusammensetzen bereits vorhandener Metriken). Hierfür wird wieder eine Dropdown-Box verwendet.

7.3.2. Henshin-Data

Werden alle benötigten Felder des Formulars aus der ersten Seite des Wizards gefüllt, so kann der Benutzer über den „Next“-Button in die nächste Seite übergehen. Wurde auf der ersten Seite „Henshin“ als Quelle für die neue Metrik ausgewählt, so wird eine entsprechende „Henshin Data“ Seite eingeblendet (Abb. 7.12).

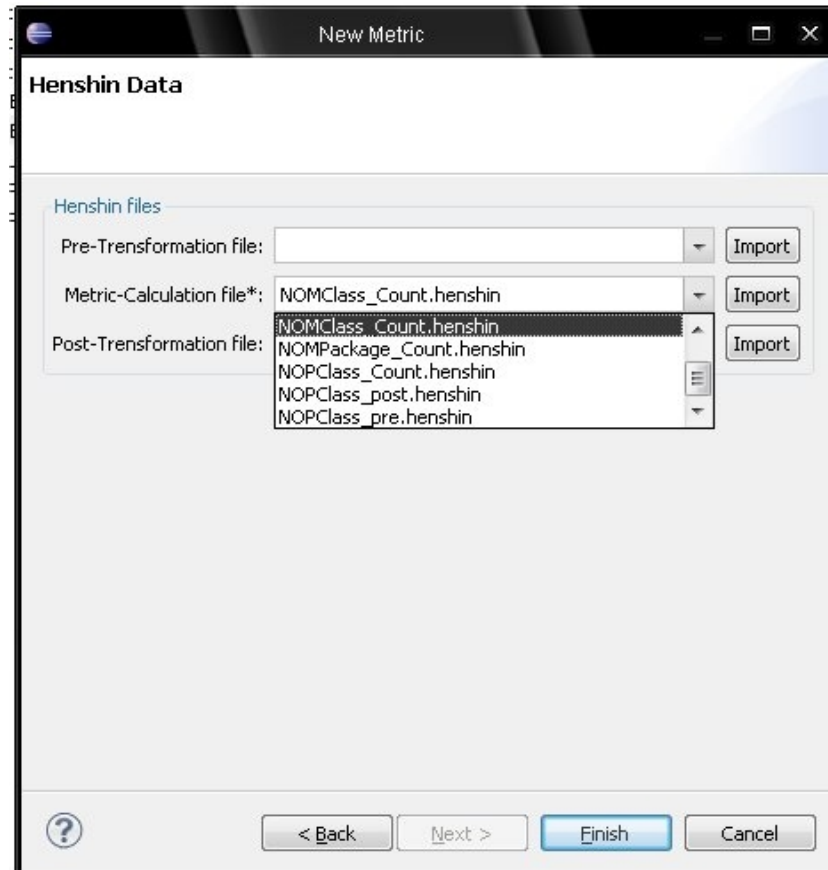


Abbildung 7.12. Screenshot: New-Metric Wizard: Henshin Data.

Das Formular auf der „Henshin Data“ Seite beinhaltet drei Eingabefelder für die Namen von drei Dateien mit Henshin Transformationen. Die Eingabefelder wurden in der Form von Dropdown-Boxen realisiert um die Eingabe von ungültigen Dateinamen zu verhindern

Die Angabe einer Regel zur Metrikenberechnung ist erforderlich um den Spezifikationsprozess erfolgreich abzuschließen. Eine Vorbereitungsregel sowie eine Regel zur Zurückerstellung der ursprünglichen Form des Modells sind optional.

Die Dropdowns werden mit Daten aus dem „/transformations“ Verzeichnis des Zielprojektes gefüllt. In diesem Beispiel wurden einige Metrikdateien in das entsprechende Verzeichnis bereits importiert (Abb. 7.13).

Sollten weitere im Projekt nicht vorhandene Transformationen gewünscht werden, so können diese über die jeweiligen „Import“-Buttons ausgewählt werden. Nach dem klicken eines „Import“-Buttons erscheint ein Dateidialog Fenster in dem der Benutzer die Möglichkeit hat die zu importierenden Dateien anzugeben. Diese werden Anschließend in das „./transformations“ Verzeichnis des Projektes kopiert. Sollte im Projekt kein Verzeichnis mit diesem Namen vorhanden sein, so wird ein erstellt.

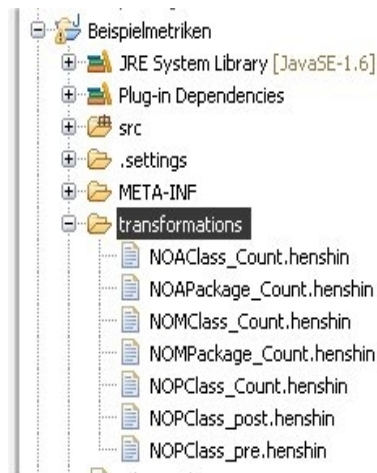


Abbildung 7.13. Screenshot: Quellordner aus dem die Henshin Dateien geladen werden.

Achtung: Der Name von dem Verzeichnis ist festgelegt und darf nicht verändert werden. Die neu erstellten Metriken beziehen sich auf die bei der Spezifikation angegebenen Henshin Dateien. Letzteren müssen in dem „./transformations“ Verzeichnis vorhanden bleiben um die Funktionalität der Metriken zu gewährleisten.

7.3.3. Composite-Daten

Wurde auf der ersten Seite „Composite“ als Quelle für die neue Metrik ausgewählt, so wird eine entsprechende „Composite Data“ Seite eingeblendet (Abb. 7.14).

Das Formular auf der „Henshin Data“ Seite beinhaltet:

- zwei Tabellen mit jeweils einer Auflistung von Metriken die zusammengesetzt werden können,
- eine Dropdown-Box mit den verfügbaren mathematischen Operationen mit denen die Metriken verbunden werden können.

Die Tabellen werden mit allen verfügbaren Metriken die dem auf der ersten Seite des Wizards angegebenen Metamodell und Kontext entsprechen.

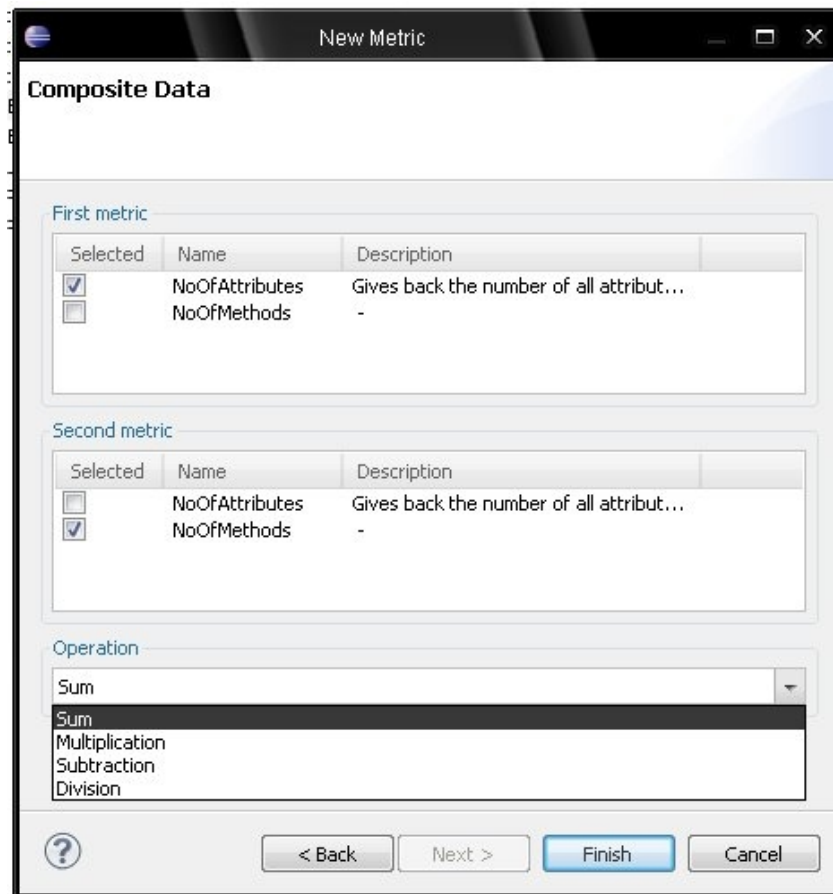


Abbildung 7.14. Screenshot: New-Metric Wizard: Composite Data

In den Tabellen werden die folgenden Attribute der Metriken angezeigt :

- Name der Metrik
- Beschreibung der Metrik

Die Angabe von zwei Metriken ist erforderlich um den Spezifikationsprozess erfolgreich abzuschließen. Es ist möglich nur jeweils eine Metrik aus jeder Tabelle auszuwählen. Metriken können mit sich selbst kombiniert werden.

Durch eine Zusammenstellung zweier Metriken entstandene Metrik kann wieder mit einer weiteren Metrik zusammengesetzt werden.

7.3.4. Generator

Nachdem alle benötigten Felder der jeweiligen zweiten Seite des Wizard gefüllt worden sind, kann der Benutzer auf den „Finish“-Button klicken und den Spezifikationsprozess beenden.

Danach werden folgende Aktionen ausgeführt:

- Eine neue Klasse wird generiert (Abb.7.16). Diese beinhaltet den Quellcode zur Metrikenberechnung. Sie wird in das Package: `de.unimarburg.swt.emf.metric` generiert, sollte das Package nicht vorhanden sein so wird ein erstellt.
- Ein Eintrag in die Datei „plugin.xml“ des Zielprojektes wird erstellt. Dieser beinhaltet Informationen über die neue Metrik und ermöglicht es auf diese über die Extension-Point Technologie zuzugreifen (Abb.7.15).

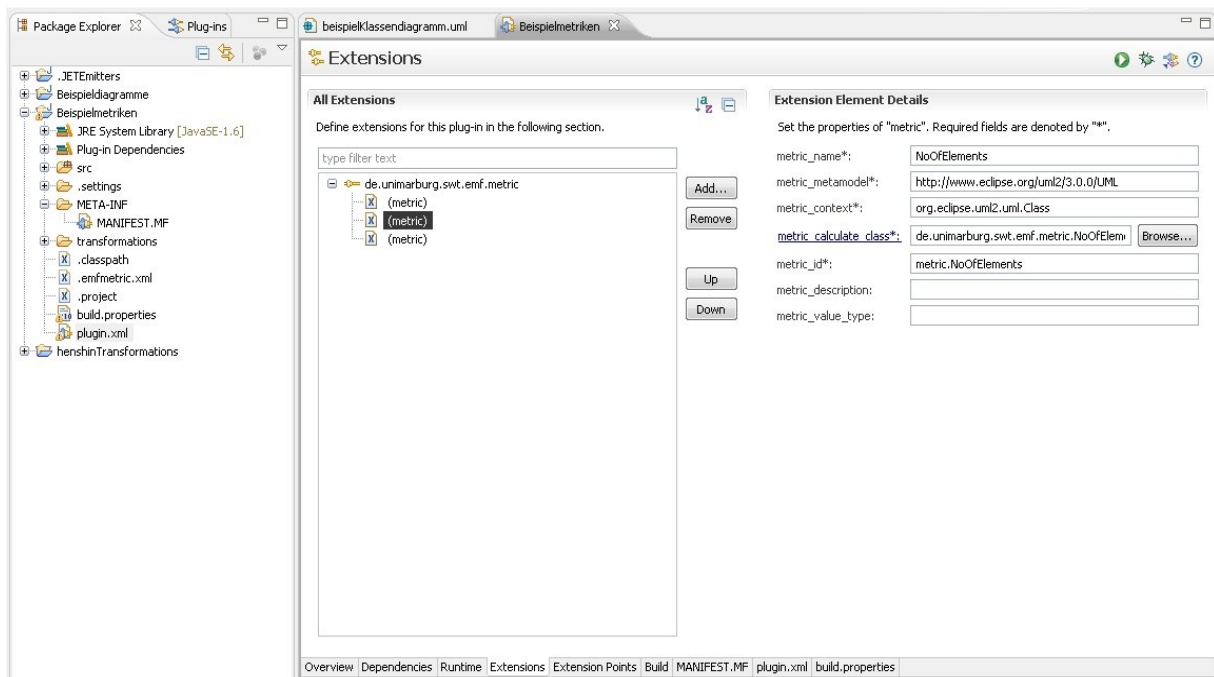


Abbildung 7.15. Screenshot: Einträge in die Datei „plugin.xml“

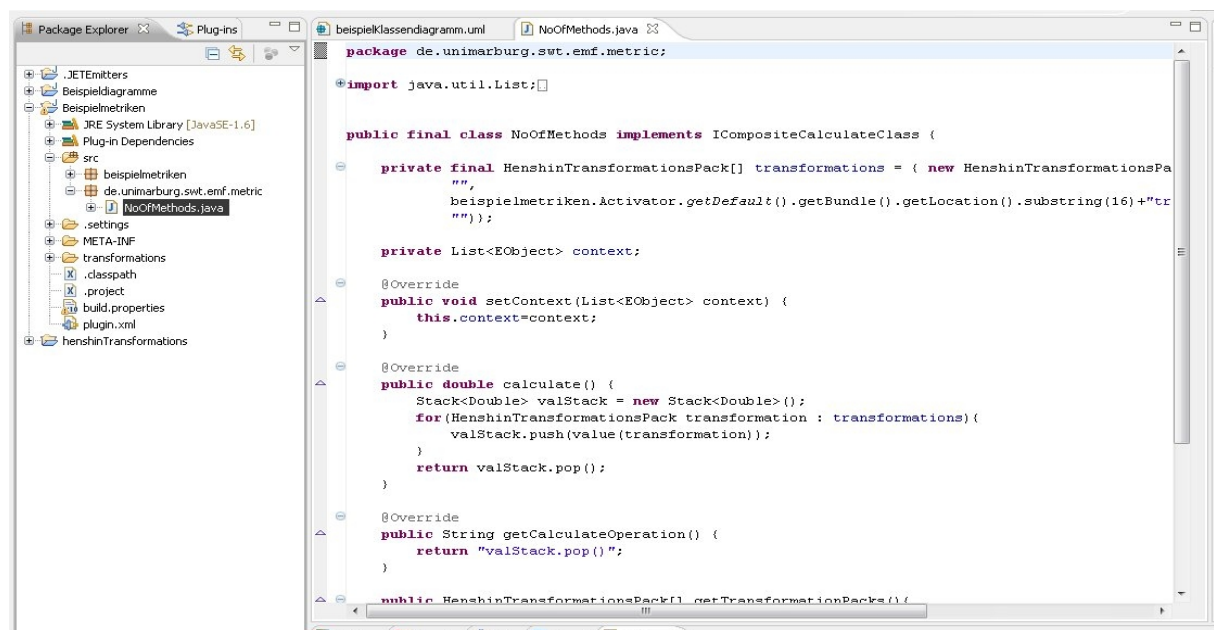


Abbildung 7.16. Screenshot: Quellcode der generierten Klasse

8. Erweiterungsvorschläge

Während der Entwicklung von „EMF Metrics“ haben sich zahlreiche neue Anforderungen aufgetaucht die zum Teil umgesetzt werden konnten. Dem zu folge ist das System aber größer gewachsen als geplant und es haben sich die folgenden Erweiterungsvorschläge herauskristallisiert:

- Die aktuelle Version könnte in zwei verschiedene Plugins aufgeteilt werden. So wurde man eine bessere Übersichtlichkeit gewährleisten. Die so entstandenen Komponente könnten einzeln genutzt oder erweitert werden.
- Das manuelle löschen einer Metrik aus einem Projekt überlässt den entsprechenden Eintrag in der Datei „plugin.xml“ des Projektes. Diese könnte jedes mal automatisch angepasst werden.
- Die Transformationsdateien werden in der aktuellen Version in Dreierpacketen behandelt. Man könnte eine neue Schnittstelle definieren die die Zusammenfassung aller drei Transformationsdateien in eine ermöglichen würde. So wurde man die durch Verweise auf nicht gültige Dateien entstehende Fehler minimieren. Das System wurde dadurch auch an Übersichtlichkeit gewinnen.
- Die Spezifikationskomponente zur Spezifikation neuer Metriken anhand von bereits vorhandenen Metriken könnte um eine Iterationsoperation erweitert werden. Momentan ist es möglich jeweils nur 2 Metriken mit einander zu verbinden. Man kann diesem Prozess natürlich beliebig oft wiederholen, aber es ist nicht möglich eine Metrik zu definieren die z.B. über alle Klassen eines Package iteriert.⁴³

⁴³ Das Problem kann man durch entsprechend definierte Henshin Transformationsdateien lösen. Dazu müssen aber neue Transformationsregeln für jede neue Iterationsmetrik definiert werden. Durch die iterationsoperation könnten bereits vorhandene Metriken zu solchen Zwecken genützt werden.

9. Ausblick

Es gibt zahlreiche Gründe zur Annahme, dass der Bedarf an Qualitätssteigenden Maßnahmen in der Softwareentwicklung steigen wird. Qualitätsanalyse in frühen Phasen der Entwicklungsprozesse ist ein immer öfter angesprochener Ansatz und erscheint als äußerst vielversprechend.

Modellmetriken bilden ein mächtiges und nützliches Werkzeug zur Qualitätsanalyse von Softwaremodellen und können somit schon in frühen Entwicklungsphasen eingesetzt werden. Der Bedarf an spezialisierten Werkzeugen die eine Durchführung solch einer Analyse ermöglichen und unterstützen werden wird demnach immer größer.

Das „EMF Metrics“ Plugin greift dieses Problem auf und bietet ein benutzerfreundliches Werkzeugpaket zur Qualitätsanalyse von Softwaremodellen. Die Verbindung zum EMF Henshin System ergab sich als sehr fruchtbar.. Henshin Transformationsregeln ermöglichen durch ihre Ausdrucksstärke die Definition von beliebigen Modellmetriken. Dabei sind sie durch den grafischen Editor unkompliziert zu erstellen.

„EMF Metrics“ wurde in der aktuellen Version als Prototyp entwickelt und bietet zahlreiche offene Schnittstellen zur Erweiterung. Eine potentielle Verbindung mit anderen sich zur Zeit in Entwicklung befindenden Systemen wie z.B „EMF Refactoring“ könnte mit einem mächtigen Werkzeugrack zur Modellbasierten Softwareentwicklung resultieren. Metriken könnten zum Beispiel zum finden von „Modell Smells“⁴⁴ benutzt werden. Metriken könnten auch anschließend vor und nach der Durchführung von einem geeignetem Refactoring berechnet werden um die Effizienz des letzteren zu messen.

⁴⁴ ein Konstrukt, das eine Überarbeitung des entsprechenden Modells nahelegt.

10. Abbildungsverzeichnis

Abbildung 2.1. Henshin-Transformationsregel	8
Abbildung 3.1. Zeitplan für das EMF Metrics Projekt	10
Abbildung 4.1. Anwendungsfalldiagramm: EMF Metrics	21
Abbildung 4.2. Aktivitätsdiagramm: Metrik spezifizieren	23
Abbildung 4.3. Aktivitätsdiagramm: Metriken konfigurieren	24
Abbildung 4.4. Aktivitätsdiagramm: Metriken berechnen	25
Abbildung 5.1. EMF Metrics Systemarchitektur	43
Abbildung 5.2. EMF Metrics Klassendiagramm	46
Abbildung 5.3. EMF Metrics Klassendiagramm (Komponenten)	47
Abbildung 5.4. EMF Metrics Klassendiagramm (MVC)	48
Abbildung 5.5. Loader-Komponente	49
Abbildung 5.6. Calculator-Komponente	50
Abbildung 5.7. Generator-Komponente	52
Abbildung 5.8. ICalculateClass Schnittstelle	53
Abbildung 6.1. getConfiguration() Aktivitätsdiagramm	59
Abbildung 6.2. Metriken konfigurieren Sequenzdiagramm	76
Abbildung 6.3. Metrik berechnen Sequenzdiagramm	77
Abbildung 6.4. Metrik spezifizieren Sequenzdiagramm	79
Abbildung 7.1. Screenshot: Properties.	83
Abbildung 7.2. Screenshot: Properties-Menü.	84
Abbildung 7.3. Screenshot: Kontextmenü vom baumbasierten UML-Editor.	85
Abbildung 7.4. Screenshot: Metrics-View.	86
Abbildung 7.5. Screenshot: Metrics-View Kontextmenü.	87
Abbildung 7.6. Screenshot: „Save results“ Dialogfenster.	87
Abbildung 7.7. Screenshot: Inhalt der gespeicherten Datei.	88
Abbildung 7.8. Screenshot: File-Menü.	89
Abbildung 7.9. Screenshot: New-Dialogfenster.	89
Abbildung 7.10. Screenshot: Kontextmenü vom baumbasierten UML-Editor.	90
Abbildung 7.11. Screenshot: New-Metric Wizard: Metric Data.	91
Abbildung 7.12. Screenshot: New-Metric Wizard: Henshin Data.	92
Abbildung 7.13. Screenshot: Quellordner aus dem die Henshin Dateien geladen werden.	93
Abbildung 7.14. Screenshot: New-Metric Wizard: Composite Data	94
Abbildung 7.15. Screenshot: Einträge in die Datei „plugin.xml“	95
Abbildung 7.16. Screenshot: Quellcode der generierten Klasse	95

11. Literatur

[BoRuJa99] Grady Booch, James Rumbaugh, Ivar Jacobson „The Unified Modeling Language“, 1999 Addison Wesley Longman.

[HaMa01] Dick Hamlet, Joe Maybee „The Engineering of Software. Technical Foundations for the Individual“, 2001 Addison Wesley Longman.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides „Design Patterns Elements of Reusable Object-Oriented Software“, 1995 Addison-Wesley.

[ClRu06] Eric Clayberg, Dan Rubel „Eclipse Building Commercial-Quality Plug-ins“ 2ed, 2006 Pearson.

[GMPG] Marcela Genero, Esperanza Manso, Mario Piattini, Francisco Garcia „Early metrics for object oriented information systems“

[Rei] Ralf Reißing „Towards a Model for Object-Oriented Design Measurement“ Stuttgart

[OMG02] Object Management Group. MDA-The OMG Model Driven Architecture, 2002

[GePiCa] Marcela Genero, Mario Piattini, Coral Calero „Early measures for UML class diagrams“ University of Castilla

[Wiki01] [http://de.wikipedia.org/wiki/Teile_und_herrsche_\(Informatik\)](http://de.wikipedia.org/wiki/Teile_und_herrsche_(Informatik)) Stand: 11.07.2001

[EMF] <http://www.eclipse.org/modeling/emf/> Stand: 11.07.2001

[Hen] <http://www.eclipse.org/modeling/emft/henshin/> Stand: 11.07.2001

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Alle wörtlich oder sinngemäß den Schriften anderer entnommenen Stellen habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt auch für beigelegte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.