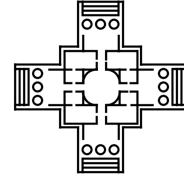
CARL VON OSSIETZKY universität OLDENBURG



Diplomarbeit

Entwicklung und Transformation eines EMF-Modells des Palladio Komponenten-Meta-Modells

9. Mai 2006

Bearbeitet von:
Klaus Krogmann[1]
Reinekeweg 2
26676 Harkebrügge
kelsaka@gmx.de

Betreut von:
Erstgutachter: Prof. Dr. Ralf Reussner[1,2]
Zweitgutachter: Prof. Dr. Wilhelm Hasselbring[1]
Betreuer: Dipl.-Wirtsch.-Inform. Steffen Becker[1,2]

[1]Fk. II, Department für Informatik, Abt. Software Engineering
an der Carl-Von-Ossietzky Universität, Oldenburg
[2]Fakultät für Informatik, IPD Reussner
an der Universität Karlsruhe (TH)

Zusammenfassung

Das Palladio Komponentenmodell stellt ein Meta-Modell für Architekturen aus Software-Komponenten zur Analyse nicht-funktionaler Eigenschaften dar. Das aktuelle Palladio Komponenten-Meta-Modell wurde bis dato in keinem Paper vollständig beschrieben. Diese Arbeit beschäftigt sich mit den Konzepten des Palladio Komponenten-Meta-Modells, sowie damit verbundenen Einschränkungen und Problemen. Darüber hinaus wurde eine Modellierung des Palladio Komponenten-Meta-Modells über eine Repräsentation in UML2 durchgeführt. Eine Diskussion der gewählten Modellierung befasst sich unter anderem mit bestehenden Entwurfs-Alternativen.

Eine Validierung der Modellierung wurde über eine Fallstudie in einem modellgetriebenen Prozess durchgeführt. Unter Verwendung des *Eclipse Modelling Frameworks* und des *Graphical Modelling Frameworks* wurde über Transformations- und Generierungsschritte ein graphischer Editor für Instanzen des Palladio Komponenten-Meta-Modell erzeugt. Im Zentrum der Untersuchung standen die Überprüfung der Eignung des erzeugten Palladio Komponenten-Meta-Modells, die Bewertung der verwendeten Werkzeuge, die Prüfung des verwendeten modellgetriebenen Prozesses und ein Vergleich mit der konventionellen (nicht modellgetriebenen) Entwicklung eines graphischen Editors für Instanzen des Palladio Komponenten-Meta-Modells.

Trotz vorhandener Mängel in den eingesetzten Werkzeugen, konnten alle Modellierungs-, Transformations- und Generierungsschritte vollständig durchgeführt werden, um in einem modellgetriebenen Prozess von einem Meta-Modell zu einem prototypischen graphischen Editor zu gelangen, der es erlaubt, die Instanzen des Palladio Komponenten-Meta-Modells zu bearbeiten.

Für meine Frau Taalke.

Inhaltsverzeichnis

1. Einleitung **11**
1.1. Idee . . . 11
1.2. Verwandte Arbeiten . . . 12
1.2.1. Komponentenmodelle . . . 12
1.2.2. Modellgetriebenen Entwicklung . . . 12
1.3. Überblick . . . 13
1.4. Meta-Modellierung . . . 14
1.4.1. UML 2.0 . . . 14
1.4.2. Komponenten-Meta-Modell . . . 15
1.5. Bemerkungen . . . 16

2. Das Palladio Komponentenmodell **17**
2.1. Idee . . . 17
2.2. Einführung . . . 18
2.3. Einfache Komponenten . . . 21
2.4. Schnittstellen und Rollen . . . 21
2.5. Kurzeinführung: Kontext . . . 23
2.6. Assembly Konnektoren . . . 23
2.7. Zusammengesetzte Komponenten . . . 25
2.8. Schnittstellen, Signaturlisten und Protokolle . . . 27
2.8.1. Signaturlisten und Signaturen . . . 27
2.8.2. Protokolle . . . 28
2.9. Service Effekt Spezifikation . . . 29
2.9.1. Parametrisierte Verträge . . . 32
2.9.2. Quality of Service . . . 32
2.10. Komponenten: Typ-Ebenen . . . 33
2.10.1. Typ-Hierarchie . . . 33
2.10.2. Komponenten-Typen . . . 34
2.10.3. Darstellung und Ebenenbeziehungen . . . 35
2.10.4. Quantitäten der Sub-Typ-Beziehungen . . . 38
2.11. Interoperabilität . . . 39
2.11.1. Substitution von Komponenten . . . 39
2.11.2. Co- und Contra-Varianz . . . 40
2.11.3. Delegations-Konnektoren . . . 42
2.11.4. Einschränkungen . . . 42
2.12. First Class Entities . . . 42
2.13. Benutzer-Rollen . . . 43
2.14. Allokation und Ressourcen . . . 43

2.14.1. Assembly 45
2.14.2. Vergleich von Allokation und Assembly 46
2.14.3. Assembly und Composite Component 47
2.15. Komponenten-Typ vs. Komponenten-Verwendung 48
2.16. Kontext 49
2.17. Hierarchie-Ebenen 52
2.18. Identität 52
2.19. Annotationen 53
2.20. Validierung von Modellinstanzen 55
2.21. Einschränkungen 56
2.21.1. Zustandsmodellierung 57
2.21.2. Mehrfache Schnittstellen-Verwendung 58
2.21.3. Rekursion von Composite Components 61
2.21.4. Stimulus-Response-Mechanismus 62
2.21.5. Proaktivität 62
2.21.6. Benutzerinteraktion 63
2.21.7. Dynamische Allokation, dynamischer Kontext 63
2.21.8. Versionierung von Komponenten 64
2.21.9. Semantikdefinition 64
2.22. Ausblick auf die Entwicklung 64
2.23. Anmerkungen 65
2.23.1. Auflösen von Composite Components 65
2.23.2. Schlechte Modellierung von Schnittstellen 65

3. Modellierung **67**
3.1. Modell Driven Architecture 67
3.2. Entwicklungswerkzeuge 69
3.3. Entwicklungsprozess 70
3.4. Anforderungen 72
3.5. UML2-Modell 72
3.5.1. Transformation von UML2 zu EMF/Ecore und Java 73
3.5.2. Grundsätze 75
3.5.3. Sub-Modelle 75
3.5.4. Modellierung 80
3.5.5. OCL-Constraints 90
3.5.6. Transformation von OCL-Constraints 100
3.5.7. Einschränkungen durch die Verwendung von RSA 100
3.5.8. Weitere Einschränkungen bei der UML2-Modellierung 102

4. Eclipse Modeling Framework **105**
4.1. Import von Serialisierungs-Artefakten 105
4.1.1. UML2-Format 105
4.1.2. ECORE-Format 106
4.2. Transformationen beim Import 106
4.2.1. Erzeugung von Java-Code 107
4.2.2. Berücksichtigung von Constraints unter EMF 108
4.2.3. Merge bestehender Modelle unter EMF 110
4.3. Verwendung von Identifiern 111

4.3.1. Alternative Modellierung von Identifiern 112
4.3.2. Umgesetzte Implementierung für Identifier 113
4.4. Testfall zur Evaluation 114

5. Graphical Modeling Framework – Editor 117
5.1. Einführung in GMF 118
5.2. Erstellungsprozess 120
5.3. Realisierung 120
5.3.1. Graphische Repräsentation 121
5.3.2. Tool-Definition 121
5.3.3. Mapping-Definition 121
5.3.4. Sichten 121
5.3.5. Umgesetzte Konzepte 124
5.3.6. Probleme und Einschränkungen 127
5.4. Fazit – GMF 133
5.4.1. Aufwand zur Nachpflege bei Modelländerungen 133
5.4.2. Bewertung von GMF 134

6. Fazit 137
6.1. Zeitplanung 137
6.2. Bewertung des MDA-Ansatzes 139
6.2.1. Informationsergänzungen 139
6.2.2. Transformationsschritte 140
6.2.3. Werkzeugunterstützung 142
6.3. Zielvergleich 143
6.4. Ausblick 143
6.5. Zusammenfassung 144

A. Anhang 147
A.1. Komponentenarten 147
A.2. Berechnung von SEFFs und Protokollen 148
A.2.1. Berechnung von benötigten Protokollen 148
A.2.2. Berechnung der SEFFs von Composite Components 149
A.3. Erstellungsanweisungen 150
A.3.1. EMF 150
A.3.2. GMF 151
A.4. UML-Diagramme 155
A.5. Abbildungen: Alternative Identifier-Modellierung 166
A.6. Glossar 168
Abbildungsverzeichnis 169
Literaturverzeichnis 174
A.7. Selbstständigkeitserklärung 181

1. Einleitung

Modellgetriebene Entwicklung [20] verspricht auf Basis abtrakter Software-Modelle, die beispielsweise in einer Notation wie der *Unified Modelling Langage* vorliegen, automatisiert kompilierbaren Software-Quellcode erzeugen zu können. Dadurch könnten Änderungen am Software-Modell ins kürzester Zeit in neuen Programmversionen resultieren und umgekehrt. Durch die Erhöhung des Grades der Automatisierung bei der Erzeugung von Software-Quellcode könnten Fehler minimiert werden. Über die Trennung von Software-Modell und generiertem Software-Quellcode soll sich zusätzlich eine Plattformunabhängigkeit des Software-Modells erreichen lassen. Insgesamt soll auf diese Weise eine Steigerung der Effizienz von Software-Entwicklungsprozessen erreicht werden.

Die Möglichkeiten der modellgetriebenen Entwicklung, auch unter dem Akronym MDA (*Modell Driven Architecture*) bekannt, stehen und fallen dabei mit der Mächtigkeit der verwendeten Werkzeuge. Je mehr Programmieraufwand durch Werkzeuge abgenommen wird, desto schneller kann die Entwicklung neuer Versionen erfolgen.

Die Stärken von MDA werden vor allem in der permanenten Synchronisation zwischen Software-Modell und Software-Quellcode gesehen. Die Abstraktion von Software-Quellcode in Form eines Software-Modells bleibt stets konsistent zur Ausprägung als Software-Quellcode und umgekehrt – ein Zugriff auf die weniger komplexe Abstraktion des Software-Quellcodes bleibt damit permanent bestehen.

Die Ausgangsbasis für eine modellgetriebene Entwicklung (in „Vorwärtsrichtung“) bildet dabei ein Domänenmodell, dass die Modell-Elemente der Anwendungsdomäne beschreibt. Da nicht Instanzen von Modellen der Anwendungsdomäne beschrieben werden, sondern Modelle gültiger Modell-Instanzen, handelt es sich bei Domänenmodellen um Meta-Modelle.

1.1. Idee

Im Rahmen der Diplomarbeit soll ein Meta-Modell des Palladio Komponentenmodells beschrieben und entwickelt werden. Das Palladio Komponenten-Meta-Modell dient im Folgenden als Domänenmodell für die Fallstudie eines MDA-Prozesses, bei dem die Verwendung aktueller Transformations-Werkzeuge zur Erzeugung eines graphischen Editors erprobt wird. Im Zentrum der Arbeit stehen die Konzepte und die Modellierung des Palladio Komponenten-Meta-Modells.

Die Durchführung der Fallstudie erfolgt im Wesentlichen anhand der Werkzeuge *Eclipse Modelling Framework* und *Graphical Modelling Framework*, um damit auch neue Entwicklungen im Bereich der MDA zu berücksichtigen. Betrachtet werden dabei die Möglichkeiten, Probleme und Einschränkungen modellgetriebener Entwicklung.

Die wesentlichen Beiträge dieser Diplomarbeit bestehen damit

- in der Bearbeitung der Konzepte des Palladio Komponentenmodells und Meta-

Modellierung

- sowie der Durchführung eines beispielhaften MDA-Prozesses zur Erzeugung eines graphischen Editors für das Palladio Komponentenmodell zur Evaluation, Validation und Bewertung des erstellten Meta-Modells und der MDA-Ansatzes.

1.2. Verwandte Arbeiten

Werden im weiteren Verlauf der Diplomarbeit Berührungspunkte mit anderen Arbeiten festgestellt, so werden diese direkt an den entsprechenden Stellen aufgezeigt.

1.2.1. Komponentenmodelle

Einen Überblick über verschiedene derzeit existierende Komponentenmodell geben Lau und Wang in [46]. Sie erstellen dabei eine Taxonomie für zwölf Komponentenmodelle und grenzen diese gegeneinander ab. Die Einordnung erfolgt unter anderem in Kategorien wie „Komponenten-Syntax“, „Komponenten-Semantik“ und „Kompositionsbegriff“.

Das *Fractal Component Model* [18] ist ein unter anderem bei der France Telecom entwickeltes Komponentenmodell. Konzeptionell sind beispielsweise zusammengesetze Strukturen für Komponenten möglich. Es zeichnet sich durch eine verfügbare vollständige Implementierung, die bereits in der industriellen Praxis erprobt ist.

Das SOFA Komponentenmodell [27] ist ein hierachisches Komponentenmodell mit einem Fokus auf funktionale Eigenschaften, die auch die Prüfung auf Protokoll-Interoperbilität einbezieht. Zudem sind Analysen auch auf parallelen Kontrollflüssen möglich. Eine Implementierung zu SOFA existiert auf Basis von Fractal.

Mit KLAPER [36] gibt es schließlich ein Meta-Modell zur Analyse von Performanz- und Zuverlässigkeitseigenschaften für komponentenbasierte Systeme – welches jedoch kein Komponenten-Modell im engeren Sinne ist.

1.2.2. Modellgetriebenen Entwicklung

Friedemann Ludwig und Frank Salger von der sd&m AG haben „Werkzeuge zur domänenspezifischen Modellierung“ [48] untersucht. Sie unterscheiden dabei einen „reinen modellgetriebenen Ansatz“ (mit der Verwendung von UML-Profilen), Eclipse Modeling Framework und Microsoft-DSL (Domain Specific Language) *Tools* voneinander. Ihr Fokus liegt auf der Betrachtung von Konzepten der Domänenmodellierung und dem Einfluss der Werkzeugwahl auf den MDA-Prozess. Beide Autoren schildern ihre Erfahrungen aus der Praxis der Entwicklung betrieblicher Informationssysteme.

Die Möglichkeiten Werkzeuge zur modellgetriebenen Entwicklung der Eclipse- und Microsoft-Welt miteinander kombinieren zu können, wird von Bézivin et al in [22] untersucht. Im Zentrum der Betrachtung stehen dabei die Transformationen zwischen Microsofts DSL-Tools und EMF auf den verschiedenen Meta-Ebenen. Insbesondere wird dabei auf die unterschiedlichen Konzepte der Meta-Modellierung in beiden Welten eingegangen. Das Paper liefert damit einen guten Vergleich der Ideen von ECORE und DSL.

Einen zum Eclipse Modeling Framework alternativen Ansatz stellen Wada et al in [75] vor. Sie offerieren ein eigenes Transformationsframework, das die Repräsentation eines

Domänenmodells in UML erlaubt und über Transformationen in attribut-orientierte Programme übersetzt. Unter attribut-orientierter Programmierung verstehen sie dabei die Möglichkeit Programmstrukturen über vorgegebene *Features* anpassen zu können. In ihrem Paper zeigen sie die Ideen und Besonderheiten von Transformationen auf, die sich dieser Form der Programmdefinition bedienen.

1.3. Überblick

Im weiteren Verlauf der Einleitung wird eine kurze Einführung in die Meta-Modellierung – auch im Bezug auf das Palladio Komponenten-Meta-Modell – sowie eine kurzer Überblick über verwandte Arbeiten gegeben.

Kapitel 2 befasst sich mit dem Palladio Komponenten-Meta-Modell, seinen Ideen und Konzepten. Dazu wird zunächst ein Überblick über das gesamte Modell gegeben, worauf eine Betrachtung aller Konzepte und Fähigkeiten folgt. Das Kapitel schließt mit bestehenden Einschränkungen und einem Ausblick auf mögliche zukünftige Entwicklungen des Palladio Komponenten-Meta-Modells.

Nachdem die theoretischen Aspekte des Palladio Komponenten-Meta-Modells dargestellt wurden, behandelt Kapitel 3 die Meta-Modellierung des Modells. Eine Einführung in die *Model Driven Architecture* (MDA) verdeutlicht die Ideen des Entwicklungsprozesses, der der Modellentwicklung dieser Diplomarbeit zu Grunde liegt. Nachfolgend werden die verwendeten Technologien dargestellt. Das Kapitel schließt mit einer ausführlichen Beleuchtung der in *Unified Modelling Language* 2.0 erstellten Repräsentation des Palladio Komponenten-Meta-Modells und geht dabei auch auf die definierten *Constraints* ein.

In Kapitel 4 wird die Verwendung des *Eclipse Modelling Frameworks* in der Diplomarbeit beschrieben. Hier geht es vor allem um die durchgeführten Transformationen sowie Besonderheiten, die sich im Bezug auf das Palladio Komponenten-Meta-Modell ergaben. Zuletzt wird ein umfassender Testfall für das Palladio Komponenten-Meta-Modell vorgestellt.

Um die Ideen der MDA an einem Fallbeispiel zu überprüfen, und um die Weiterverwendung des erstellten Palladio Komponenten-Meta-Modell zu überprüfen, wurde ein graphischer Editor prototypisch mit Hilfe des Graphical Modelling Frameworks (GMF) erstellt. Kapitel 5 beschreibt zunächst die Funktionalität und Komponenten von GMF. Es schließt sich eine Erläuterung der durchgeführten Realisierung des graphischen Editors an und endet mit einer Bewertung von GMF vor dem Hintergrund der Ideen der MDA. Dabei wird auch ein Vergleich mit einem alternativen graphischen Editor des Palladio Komponenten-Meta-Modells gezogen.

Jedes einzelne Kapitel endet mit einem Fazit für den jeweils behandelten Themenbereich. Das Fazit in Kapitel 6 bewertet schließlich den in der Diplomarbeit durchgeführten MDA-Prozess in seiner Gesamtheit. Dazu werden die Ziele der Diplomarbeit (vgl. [44]) mit der Umsetzung verglichen und die Zeitplanung in der Retrospektive betrachtet. Die vorliegende Diplomarbeit endet mit einem Ausblick auf zukünftige Entwicklungen.

1.4. Meta-Modellierung

Die begriffliche Unterscheidung von Modell-Ebenen kann im Bereich der Meta-Modellierung schnell unscharf werden. Insbesondere durch Meta-Meta-Ebenen, bei denen sich das „Meta“-Präfix häuft, fällt die Unterscheidung der Meta-Ebenen schwer. Abbildung 1.1 verdeutlicht aus diesem Grund die Modell-Ebenen, wie sie für die Modellierung des Komponenten-Meta-Modells zutreffen und vergleicht sie mit den Modell-Ebenen der UML 2.0. Zusätzlich wird ein Beispiel für jede Modell-Ebene gegeben.

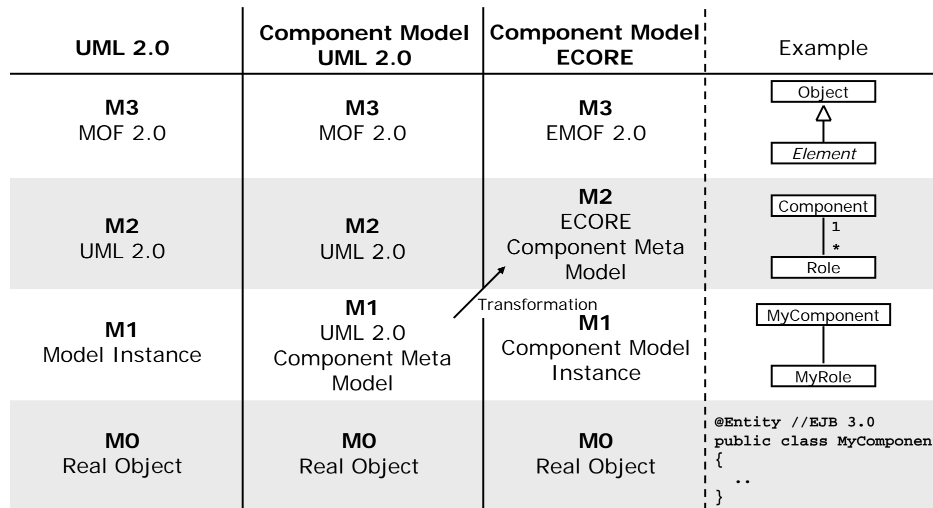


Abbildung 1.1.: Meta-Modell-Ebenen der UML 2.0 und des Komponenten-Meta-Modells im Vergleich

Eine sinnvolle Meta-Modellierung setzt naheliegender Weise mindestens zwei Modell-Ebenen voraus. Auf der einen Ebene liegt die Instanz eines Modells, auf der anderen Ebene liegt das Meta-Modell der Instanz. Alle Erweiterungen um zusätzliche Meta-Ebenen setzen dieses Prinzip fort. So ist zur Beschreibung des Meta-Modells eine weitere Modell-Ebene möglich. Dies ist das Meta-Modell des Meta-Modells. Aus Sicht der Instanz-Ebene handelt es sich hierbei um das Meta-Meta-Modell. In der Literatur wird häufig auf zusätzliche „Meta“-Präfixe zu Gunsten der Lesbarkeit verzichtet. Dadurch verschwimmt jedoch die Unterscheidung der Meta-Ebenen untereinander. In der vorliegenden Diplomarbeit werden daher an allen Stellen, an denen eine Unterscheidung der Modell-Ebenen von Bedeutung ist, die zusätzlichen Präfixe erhalten.

Um den Einstieg in die Meta-Modellierung zu erleichtern, sollen im Folgenden die Meta-Ebenen der UML 2.0 dargestellt werden. Da das Komponenten-Meta-Modell ebenfalls unter Rückgriff auf die UML 2.0 beschrieben wird, erscheint die Einführung angemessen.

1.4.1. UML 2.0

Die UML 2.0 [62] selbst stellt ein Meta-Modell für alle Instanzen von UML 2.0-Modellen dar. In Abbildung 1.1 (Spalte „UML 2.0“) wird UML 2.0 als „M2“-Ebene, die Instanzen von UML 2.0-Modellen als „M1“-Ebene dargestellt. UML 2.0 besitzt MOF 2.0 [65, 63] als Meta-Modell. MOF 2.0 wiederum wird durch sich selbst beschrieben. Damit besitzt MOF 2.0 sich selbst als Meta-Modell. Auf diese Weise wird eine Verlagerung auf immer

weitere Meta-Modell-Ebenen vermieden, zugleich ergibt sich ein „Henne-Ei“-Problem. Die Meta-Modellierung von MOF 2.0 muss sich selbst eindeutig ohne Rückgriff auf andere Meta-Modelle beschreiben.

Die „M0“-Ebene stellt definitionsgemäß die Objektebene dar, ist also keine Modell-Ebene. Darauf baut die „M1“-Ebene auf. Dies ist, wie bereits beschrieben, die Instanz-Ebene. Sie ist ein Modell der „M0“-Ebene. Zur Beschreibung der Menge aller möglichen Modell-Instanzen dient das Meta-Modell auf der Ebene „M2“ – hier mit UML 2.0. Das Meta-Meta-Modell zur „M1“-Ebene und das Meta-Modell zur M2-Ebene ist die „M3“-Ebene – hier in der Ausprägung mit MOF 2.0.

1.4.2. Komponenten-Meta-Modell

In einem ersten Schritt sollte das Palladio Komponenten-Meta-Modell mittels UML 2.0 beschrieben werden. UML 2.0 wurde als Modellierungssprache verwendet, da diese leicht verständlich ist und vor allem sehr gut durch Modellierungs-Werkzeuge unterstützt wird. Über einen Transformationsschritt (vgl. auch Abbildung 1.1) sollte die Beschreibung des Komponenten-Meta-Modells in UML 2.0 auf die Ebene eines ECORE Meta-Modells gehoben werden. Bei diesem Transformationsschritt werden UML-Klassen zu *EClasses* und Assoziationen zu *EReferences* des Komponenten-Meta-Modells in ECORE.

Bei der Modellierung in UML (Spalte „Component Model UML 2.0“) nimmt die UML 2.0 die Position des Meta-Modells für das Komponenten-Meta-Modell. Das Komponenten-Meta-Modell ist also eine Instanz der UML 2.0. Durch die Transformation des mit UML 2.0 beschriebenen Komponenten-Meta-Modells („M1“-Ebene) in eine EMOF-Modell-Instanz, wird das Komponenten-Meta-Modell als EMOF-Modell-Instanz (Spalte „Component Model ECORE“) zu einem Modell der „M2“-Ebene. EMOF ist dabei bis auf wenige Unterschiede identisch zu MOF umd bildet das Meta-Modell für ECORE-Modell-Instanzen.

Eine Instanz des Komponenten-Meta-Modells, beschrieben durch ein ECORE-Modell, wäre demnach wieder eine „M1“-Ebene, jedoch in der Meta-Modell-Hierarchie von „Component Model ECORE“. Anhand eines Beispiels sollen die Meta-Ebenen des „Component Model ECORE“ illustriert werden.

Für die Meta-Modellierung des Palladio Komponenten-Meta-Modells wäre eine EJB Entity Bean [73] (wie im Beispiel in Abbildung 1.1 dargestellt) eine mögliches reales Objekt der „M0“-Ebene. Im Bereich der Software ist die Entscheidung, ob Software tatsächlich das reale Objekt der Ebene „M0“ darstellt, schwer. Da Software selbst abstrakt ist und zumeist ein Modell realer Systeme – beispielsweise der Geschäftsabläufe einer Bank – darstellt, sprechen Argumente dafür, dass es sich bei Software um eine Meta-Ebene handeln müsse. Zusätzlich kann Software (im Form von Quellcode) als Meta-Modell für Laufzeitinstanzen oder Speicherzustände angesehen werden. Eine eindeutige Einordnung in absolute Ebene ist daher nicht möglich.

Im Rahmen der Meta-Modellierung stellte Software in Form von Quellcode die konkreteste, sinnvollerweise zu betrachtende Ebene dar. Entsprechend wird diese Ebene als „M0“-Ebene angesehen.

Das Beispiel aus Abbildung 1.1 stellt als „M1“-Ebene eine Komponente „MyComponent“ dar, die mit einer Rolle „MyRole“ assoziiert ist. Für diese Modell-Ebene stellt das Komponenten-Meta-Modell die Meta-Ebene „M2“ dar. Eine Komponente kann

dort `0..*` Rollen assoziieren. Auf der nächsten Ebene „M3“ stellt EMOF das Meta-Modell für das Komponentenmodell dar. Das Komponenten-Meta-Modell wird also selbst durch eine EMOF-Modell-Instanz (ECORE) beschrieben. Auf dieser Ebene tragen beispielsweise Klassen `0..*` Attribute. EMOF schließlich ist wie MOF 2.0 [65, 63] durch sich selbst beschrieben und spezifiziert. Im Beispiel wird die EMOF-Vererbungsstruktur zwischen „Element“ und „Object“ dargestellt.

Um die Modell-zu-Meta-Modell-Beziehung zwischen den Modell-Ebenen zu kennzeichnen, wird der Stereotyp „instance-of“ verwendet.

1.5. Bemerkungen

Ist im Folgenden von *Komponentenmodell* die Rede, so ist, sofern nicht anders angegeben, das *Palladio* Komponentenmodell gemeint. Als Palladio Komponentenmodell wird dabei das *Meta*-Modell zur Darstellung von Komponentenarchitekturen der Palladio-Gruppe [70] bezeichnet. Dieses Modell wird in Kapitel 2 näher beschrieben.

Einen Überblick über bestehende Komponentenmodelle liefern Lau und Wang. Sie stellen in [46, 45] eine Taxonomie für Komponentenmodelle auf und vergleichen die Konzepte verschiedener Komponentenmodelle miteinander.

Der Begriff *Projekt* bezeichnet, solange er ohne sonstigen Kontext verwendet wird, die Diplomarbeit.

2. Das Palladio Komponentenmodell

Im Rahmen der Diplomarbeit wird das Palladio Komponentenmodell erstellt. Daher ist es unabdingbar, dass ein vollständiges Verständnis für das Komponentenmodell existiert. Die folgenden Kapitel zeigen detailliert die Eigenschaften des Komponentenmodells auf. Schließlich wird dem gegenübergestellt, welche Konzepte des Komponentenmodells umgesetzt werden konnten, welche Einschränkungen bei der Modellierung auftraten und welche Anpassungen aus welchen Gründen zur Abbildung notwendig waren.

Das hier beschriebene Komponenten-Meta-Modell ist als Anforderungsdefinition für die spätere UML2- und ECORE-Repräsentation des Komponenten-Meta-Modells zu betrachten.

Das im Folgenden dargestellte Palladio Komponentenmodell fußt auf dem konzeptionellen Stand aus dem Februar 2006. Dieser beinhaltet bereits Änderung, die aus dieser Diplomarbeit erwachsen sind.

Die Modellierung des Komponentenmodells sowie die Modellierungsalternativen werden in Kapitel 3 diskutiert. Das folgende Kapitel befasst sich ausschließlich mit dem Meta-Modell.

2.1. Idee

Komponentenarchitekturen existieren in Form realer komponentenbasierter Softwaresysteme. Das Palladio Komponenten-Meta-Modell bietet eine strukturierte Form zur Erfassung solcher Komponentenarchitekturen, mithin Komponentenmodell-Instanzen. Das Komponenten-Meta-Modell versteht sich dabei als Datenspeicher, der eine Menge von Basis-*Constraints* definiert, die die Konsistenz und Integrität des Datenspeichers sichern. Damit lassen sich fest definierte Strukturen realer komponentenbasierter Softwaresysteme als Komponentenmodell-Instanzen modellieren.

Betrachtet man die gerade skizzierte Funktionalität als „Datenspeicher", so muss diese weiter unterschieden werden. Auf der einen Seite realisiert das Komponentenmodell ein Komponenten-*Reposity*, in dem Komponenten(-Typen) vorgehalten werden, auf der anderen Seite gibt es die Möglichkeit, mit Hilfe von Komponenten-Typen aus dem Repository die bereits angesprochenen Komponentenarchitekturen zu konstruieren.

Bewusst und klar von der Funktionalität eines Datenspeichers für Komponentenarchitekturen getrennt ist die Anwendung oder Definition von Algorithmen auf Komponentenmodell-Instanzen. Entsprechend der Grundsätze des Strategie Musters (*Strategy Pattern*, vgl. [34], S. 373ff), können auf einer Modell-Instanz beliebig viele Algorithmen angewendet werden, die jeweils eine eigene Strategie repräsentieren. Daher sind weitergehende Algorithmen, die Bereiche wie Validitätsprüfung, Interoperabilitätsprüfung oder ähnliches betreffen, nicht fester Bestandteil des Komponenten-Meta-Modells, sondern als optionale Elemente flexibel anwendbar.

2.2. Einführung

Die Konzepte des Komponentenmodells hängen auf vielfältige Weise von einander ab. Um einzelne Konzepte vollständig erklären zu können, ist bereits ein Wissen über andere Konzepte notwendig. Diese Konzepte verlangen wiederum nach dem Verständnis weiterer Konzepte des Komponentenmodells. Insgesamt lässt sich das Komponentenmodell damit schwer in einzelnen Ausschnitten verstehen, sondern muss als Gesamtmodell verstanden werden.

Um den Einstieg in die Konzept des Komponentenmodells zu erleichtern, widmen sich die nächsten Abschnitte einer groben Beschreibung der Elemente des Komponentenmodells. Eine detaillierte Betrachtung erfolgt dann im Anschluss.

Komponenten und Schnittstellen *Komponenten* sind Kernelemente des Komponentenmodells. Sie stellen eine Software-Einheit dar, die funktionale und nicht-funktionale Eigenschaften bündelt. Nach außen werden sie über *Schnittstellen* charakterisiert. Die Verbindung zwischen Schnittstellen und Komponenten wird über so genannte *Rollen* hergestellt. Je nachdem ob eine Schnittstelle von einer Komponente angeboten (*provided*) oder benötigt (*required*) wird, heißt die Rolle *Provided Role* beziehungsweise *Required Role*.

Für Komponenten bedeutet dies, dass sie Aufrufe auf ihren angebotenen Schnittstellen annehmen müssen (einen Dienst erfüllen müssen), wobei sie selbst Aufrufe auf den ihren benötigten Schnittstellen durchführen dürfen, um den eigenen Aufruf zu bearbeiten.

Komponenten selbst werden unterschieden in *Komponenten-Typen* auf der einen Seite und *Komponenten-Realisierungen* auf der anderen Seite. Insgesamt wird durch diese Formen von Komponenten eine Hierarchie definiert. Komponenten-Typen unterscheiden sich in *Provided Komponenten Typ*, *Complete Komponenten Typ* und *Implementation Komponenten Typ*. Hinter dieser Unterscheidung steht die Idee einer schrittweisen Verfeinerung der Menge der Informationen über Komponenten. Während *Provided Komponenten Typen* wenige Aussagen zulassen, weil sie wenig Informationen enthalten, ergänzen *Implementation Komponenten Typen* alle Informationen, die nach dem Verständnis des Komponentenmodells für einen Komponenten-Typ möglich sind.

Zu Komponenten-Typen gibt es Komponenten-Realisierungen. Diese erlauben es, Informationen über das *Innere* einer Komponente zu erfassen. Sie kommen in zwei Ausprägungen vor: auf der einen Seite als *Composite Component* und auf der anderen Seite als *Basic Component*. Nach außen (über die Schnittstellen) lassen sich beide Formen der Komponenten-Realisierung nicht von einander unterscheiden. Nach innen sind beide Formen jedoch unterschiedlich aufgebaut.

Composite Components erlauben die Zusammenfassung einer beliebigen Menge von Komponenten (egal ob Komponenten-Typen oder Komponenten-Realisierungen) zu neuen Komponenten. Sie stellen zusammengesetzte Komponenten dar. Das Verhalten von *Composite Components* resultiert vollständig aus dem Verhalten von inneren Komponenten.

Im Gegensatz dazu stehen *Basic Components*. Ihr Verhalten (in funktionaler und nicht-funktionaler Hinsicht) resultiert aus keinen weiteren internen Komponenten, sondern aus der *Basic Component* selbst. Um das Verhalten von *Basic Components* näher charakterisieren zu können, existiert die *Service Effect Specification* (SEFF). Diese gibt

in abstrahierter Form an, wie sich einzelne Dienste einer *Basic Component* verhalten.

Signaturen Bisher wurden Schnittstellen noch nicht näher betrachtet. Auf jeder Schnittstelle lassen sich *Signaturen* und *Protokolle* definieren. Signaturen entsprechen dabei Diensten, vergleichbar mit Methoden einer Klassen. Sie bestehen wie eine Methoden-Signatur aus Rückgabewert, Signaturname, Parametern und Ausnahmen (*Exceptions*). Protokolle hingegen geben an, welche Aufrufreihenfolgen von Signaturen gültig sind; etwa, als regulärer Ausdruck niedergeschrieben, $(aab)^*$, wobei a und b Signaturnamen sind.

Kontext Mit den bislang dargestellten Konzepten lassen sich noch keine Komponenten-Architekturen, also „Verdrahtungen von Komponenten untereinander, um eine bestimmte Funktion zu erfüllen", realisieren. Alle bis hier dargestellten Komponenten verfügten über keine „Verdrahtung" der angebotenen und benötigten Schnittstellen nach außen. Die Komponenten stellten bislang lediglich Elemente aus dem Komponenten-*Repository* dar. Im Folgenden werden die bisher dargestellten Komponenten zur Unterscheidung als Komponenten-Typen bezeichnet.

Um Komponenten-Typen in Komponenten-Architekturen verwenden zu können, gibt es das Konzept des *Kontexts*. Dieses macht jede Verwendung eines Komponenten-Typs (als eine Art „Verwendungs-Instanz") eindeutig. Ein verwendeter Komponenten-Typ wird auch *Kontext-Komponente* genannt, um eine klare Unterscheidung zu Komponenten-Typen herbeizuführen. Damit auch die Verwendung von Rollen, die ein Komponenten-Typ inne hat, eindeutig ist, gibt es analog zu den Kontext-Komponenten *Kontext-Rollen*.

Kontexte charakterisieren damit die tatsächlich aufgerufenen Dienste eines Komponenten-Typs und die konkreten Erbringer benötigter externer Dienste. Zusätzlich hängen die von einem Komponenten-Typ verwendeten Ressourcen vom Kontext ab, manifestieren sich also erst im Kontext.

Konnektoren Um ausdrücken zu können, dass Aufrufe externer benötigter Dienste von Kontext-Komponente A auf angebotene Dienste von Kontext-Komponente B delegiert werden, gibt es *Assembly Konnektoren*. Sie verbinden genau zwei Kontext-Rollen miteinander: eine angebotene und eine benötigte. Über die Verbindung zwischen Kontext-Komponente und Kontext-Rolle ist dann eindeutig bestimmt, welche Kontext-Komponente welche andere verwendet.

Für das Innere von *Composite Components* gibt es eine besondere Form von Konnektoren: *Delegations-Konnektoren*. Sollen Aufrufe einer von einer *Composite Component angebotenen* äußeren Schnittstelle zu einer inneren Komponente der *Composite Component* geleitet werden, erfolgt dies über einen *Provided Delegations-Konnektor*. Analog erfolgen Aufrufe auf der *benötigenden* Seite der Composite Component über *Required Delegations-Konnektoren*.

Ressourcen und Allokation *Ressourcen* werden in *berechnende Ressourcen* und *nicht-berechnende (kommunizierende) Ressourcen* unterteilt und ermöglichen eine Abbildung von Komponenten-Architekturen auf Hardware. Berechnende Ressourcen können CPU, Server oder ähnliches sein – nicht-berechnende Ressourcen sind bspw. LAN- und WAN-

Verbindungen. Dabei können nicht-berechnende Ressourcen berechnende Ressourcen miteinander verbinden.

Um Ressourcen nutzen zu können, müssen Komponenten und Assembly Konnektoren zunächst allokiert werden. Die Allokation ordnet dann eine Menge von Komponenten einer berechnenden Ressource zu und eine Menge von Assembly Konnektoren nicht-berechnenden Ressourcen zu.

2.3. Einfache Komponenten

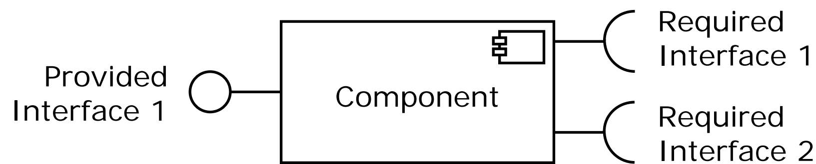


Abbildung 2.1.: Einfache Komponente mit einer angebotenen und zwei benötigten Schnittstellen

Das Palladio Komponentenmodell – unter [9] findet sich die Beschreibung einer älteren Fassung des Modells – beschreibt Software-Architekturen als eine Menge von Komponenten und Schnittstellen, sowie darauf definierten Relationen. In Abbildung 2.1 wird eine einfache Komponente „Component“ gezeigt, die eine Schnittstelle anbietet (*Provided Interface*), im Beispiel „Provided Interface 1“ genannt. Daneben benötigt die Komponente zwei Schnittstellen („Required Interface 1 / 2“).

Im Folgenden sollen zunächst einfache Basiskonstrukte des Komponentenmodell dargestellt werden. Eine detaillierte Erklärung, welche Konstrukte in welchen Situationen als valide bewertet werden, folgt in den nächsten Kapiteln (Kapitel A.2ff). In den Abbildungen dieses Kapitel wird auf die standardisierte Visualisierung mit Hilfe von UML 2 Komponentendiagrammen [39, 61, 62] zurückgegriffen.

Eine Komponente kann `0..*` Schnittstellen anbieten. Das bedeutet, dass die Komponente die Dienste auch anbieten muss, die über eine Schnittstelle festgelegt sind. Der Aufbau einer Schnittstelle wird in Kapitel 2.4 beschrieben. Für eine Komponente darf im Allgemeinen jedoch nicht angenommen werden, das angebotene Schnittstellen und darauf definierte Dienste auch tatsächlich angesprochen werden. `0` angebotene Schnittstellen sind beispielsweise denkbar, wenn eine Komponente einen Taktgeber realisiert, der sich nicht konfigurieren lässt und in festen Zeitabständen Dienstaufrufe durchführt.

Auf der *Required*-Seite kann eine Komponente `0..*` Schnittstellen benötigen. Durch die *Required*-Schnittstelle definiert eine Komponente die Dienste, die sie selbst benötigt. Insgesamt muss eine Komponente mindestens eine Schnittstelle anbieten oder benötigen. Komponenten ohne jegliche Schnittstellen könnten nicht mit anderen Komponenten interagieren und wären nicht sinnvoll.

Eine Komponente kann im Übrigen die gleiche Schnittstelle anbieten und zugleich verlangen. Stellt man sich beispielsweise eine *Chain-of-Responsibility* (vgl. [34], S. 410ff) vor, die über Komponenten realisiert wird, so wird die gleiche Schnittstelle zur Annahme (*provided*) von Dienstaufrufen und zur Weitergabe von Dienstaufrufen an den Nachfolger in der Kette verwendet.

Eine Komponente darf *eine* Schnittstelle (Gleichheit entsprechend der Identität, vgl. Kapitel 2.18) `0..1` mal anbieten und `0..1` mal benötigen.

2.4. Schnittstellen und Rollen

Definition Über Abbildung 2.2 soll verdeutlicht werden, welche Bestandteile der graphischen Notation, die auch im Folgenden verwendet wird, einem Interface entsprechen. Auf der linken Seite ist eine Komponente dargestellt, die eine Schnittstelle anbietet.

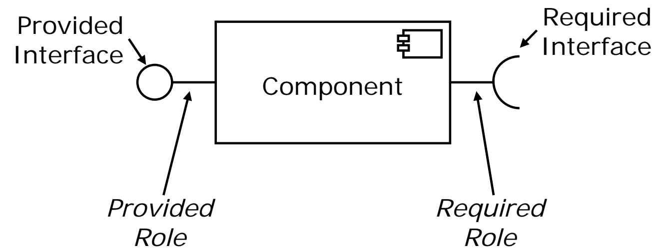


Abbildung 2.2.: Angebotene und benötigte Schnittstellen mit ihren dazugehörigen Rollen in UML2-Notation, sowie anbietende / benötigende Komponente (in abkürzender Schreibweise)

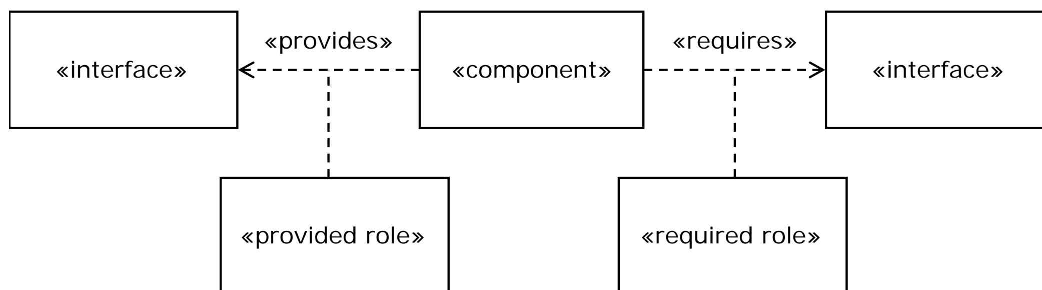


Abbildung 2.3.: Angebotene und benötigte Schnittstellen mit ihren dazugehörigen Rollen in UML2-Notation (als Schreibweise mit Stereotypen). Semantisch äquivalent zu Abbildung 2.2.

Bietet eine Komponente eine Schnittstelle an (*Provided Interface*), so impliziert dies, dass eine *Provided Role* zwischen Komponente und Interface existiert. In der graphischen Notation ist sie vergleichbar mit der Verbindungslinie zwischen Schnittstelle und Komponente. Wird eine Komponente angeboten, resultiert dies in einer Kreisdarstellung für die Schnittstelle.

Auf der rechten Seite der Abbildung ist die *Required Role* zwischen einer Komponente und der Schnittstelle dargestellt. Auch hier entspricht die Verbindungslinie zwischen Komponente und Schnittstelle der Rollen-Beziehung. Die graphische Notation ändert sich jedoch. Aus dem Kreis, der die Schnittstelle symbolisiert, wird ein Halbkreis.

Im Komponentenmodell wird die Relation zwischen Komponente und Schnittstelle als „Rolle“ (*Role*) bezeichnet. Mit Rolle wird also eine an eine Komponente gebundene Schnittstelle bezeichnet. Die Art der Rolle (angeboten oder benötigt / *provided* oder *required*) entscheidet über die Verwendung der Schnittstelle. Die Eigenschaft der Schnittstelle selbst ändert sich jedoch nicht durch ihre Verwendung. Auch in Abbildung 2.2 könnten die Komponenten der beiden dargestellten Schnittstellen identisch sein.

Eine Schnittstelle wird also erst durch eine *Provided Role* zu einer angebotene Schnittstelle / einem *Provided Interface*, bzw. über eine *Required Role* zu einer benötigten Schnittstelle / einem *Required Interface*. Die gleiche Schnittstelle kann über Rollen an mehrere Komponenten gebunden werden.

Sub-Typ-Beziehung Schnittstellen können untereinander in einer Sub-Typ-Beziehung stehen. Dabei kann eine Schnittstelle Sub-Typ mehrerer anderer Schnittstellen sein. Die Sub-Typ-Beziehung ist, analog zum Verständnis der Vererbung bei objektorientierten Programmiersprachen (vgl. [37], S. 97ff), eine Relation zwischen einer Sub-

Typ-Schnittstelle und einer Menge von Super-Typ-Schnittstellen. Die Kernsemantik der Sub-Typ-Beziehung definiert, dass eine Super-Typ-Schnittstelle durch eine ihrer Sub-Typ-Schnittstelle ersetzen werden kann.

Eine Schnittstelle kann mehrfach Super-Typ anderer Schnittstelle sein. Durch die Sub-Typ-Beziehung muss jedoch ein azyklischer Typenbaum entstehen.

Sub-Typ-Schnittstellen müssen stets die Semantik der Super-Typ-Schnittstellen erhalten. Die Einhaltung der Semantik wird nicht durch das Komponentenmodell überprüft, sondern wird, wie alle anderen Validitätsprüfungen, durch externe Validierungsalgorithmen, die auf dem Komponentenmodell arbeiten, übernommen. Die Überprüfung der Einhaltung der Semantik auf Schnittstellen durch das Komponentenmodell ist nicht sinnvoll, da die Bestimmung der Semantik unterschiedlichen Definitionen folgen kann.

2.5. Kurzeinführung: Kontext

Um die weiteren Konzepte des Komponentenmodells korrekt darstellen zu können, wird eine kurze Definition des Begriffs des *Kontexts* vorgezogen. In den nächsten Kapiteln reicht ein Basis-Verständnis des Konzepts aus. Die ausführliche Beschreibung des Kontexts findet sich in Kapitel 2.16 auf Seite 49, wo, unter Berücksichtigung der bis dahin eingeführten Hierarchie des Komponentenmodells, eine genauere Definition erfolgen kann.

Bisher wurden Komponenten und Rollen vorgestellt. Tatsächlich wurden noch keine Möglichkeiten zur Interaktion zwischen Rollen vorgestellt. Komponenten, die Rollen anbieten, werden bis jetzt also von ihrer Verwendung losgelöst beschrieben. In diesem Zusammenhang eignet sich der Begriff des *Komponenten-Typs* zur Einordnung des aktuellen Verständnisses: Welche Rollen eine Komponente einnimmt, wird über ihren Typ definiert.

Komponenten und Rollen können jedoch auch verwendet werden. Werden Komponenten und Rollen verwendet, stehen sie, vereinfacht betrachtet, in einem Verwendungs-*Kontext*. Es ist also strikt zwischen *Komponenten-Typen* und der Verwendung von Komponenten-Typen in einem *Kontext* zu unterscheiden. Ein Komponenten-Typ kann also in einer beliebigen Anzahl (`0..*`) von Kontexten verwendet werden. Der Kontext ermöglicht eine eindeutige Identifikation des Verwendungsortes. Die Verwendung einer Komponente bzw. Rolle impliziert einen Kontext, in dem die Komponente bzw. Rolle liegt. Der Kontext sollte als immanente Eigenschaft verstanden werden, die an die Verwendung von Komponenten geknüpft ist.

Um eine einfachere Unterscheidung zwischen Komponenten (als Typ) bzw. Rollen und Komponenten bzw. Rollen im Kontext zu ermöglichen, werden im Kontext verwendete Komponenten und Rollen im Folgenden kurz *Kontext-Komponenten* und *Kontext-Rollen* genannt.

2.6. Assembly Konnektoren

In Abbildung 2.4 wird die Verwendung zweier Assembly Konnektoren (*Assembly Connector*) dargestellt. Assembly Konnektoren verbinden je eine angebotene Kontext-Rolle (*Provided Role*) und eine benötigte Kontext-Rolle (*Required Role*). Aufrufe auf der benötigten Schnittstelle, die über die *Required Role* mit der Komponente verbunden

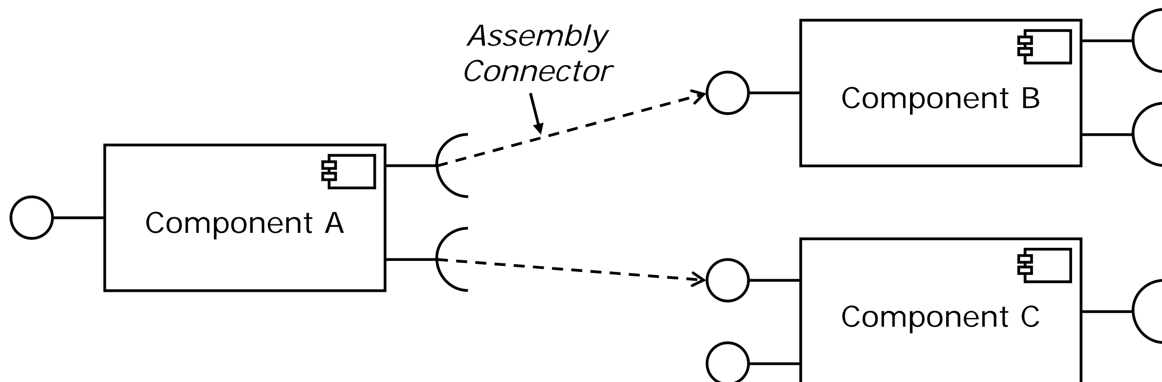


Abbildung 2.4.: Eine Komponente wird über Assembly Konnektoren mit zwei weiteren Komponenten verbunden

ist, werden auf die angebotene Schnittstelle und damit letztlich auf die Komponente, die über die *Provided Role* mit der Schnittstelle verbunden ist, delegiert.

Jede benötigte Schnittstelle einer Komponente kann damit mit genau einer angebotenen Schnittstelle verbunden werden. Sind die angebotene Schnittstelle und die benötigte Schnittstelle einer Komponente identisch oder interoperabel (zu den Definitionsmöglichkeiten von Interoperabilität siehe Kapitel 2.20), so gilt der Assembly Konnektor als valide.

Dass Assembly Konnektoren Rollen im *Kontext* miteinander verbinden und damit indirekt Komponenten im Kontext, jedoch nicht Rollen und Komponenten als Typ, wird schnell deutlich, wenn man bedenkt, dass Komponenten einen internen Zustand haben können, der von der aktuellen Verwendung (dem Kontext) abhängt.

Ein Assembly Konnektor kann lediglich eine *Required*-Schnittstelle mit einer *Provided*-Schnittstelle verbinden, die ansonsten jeweils *nicht* über Assembly Konnektoren verbunden sind. Diese Einschränkung (vgl. Kapitel 2.21.2) garantiert, dass der Kontrollfluss klar definierten Verbindungen folgt.

Der Kontrollfluss folgt im Allgemeinen der als gestrichelt dargestellten Linie in Pfeilrichtung. Der Aufruf von Methoden erfolgt von der Seite der benötigenden Komponente hin zur anbietenden Komponente. Im Falle der Verwendung von *Events* kann die Richtung des Kontrollflusses jedoch auch abweichen. Während der Kontrollfluss bei der Registrierung eines *Event*-Delegaten noch von benötigender Komponente zu anbietender Komponente geht, erfolgt der Rückaufruf (Aufruf eines Delegaten; *Call-Back*) in entgegengesetzter Richtung. Die anbietende Komponente ruft den Delegaten auf der benötigenden Komponente auf. Eine explizite Modellierung von *Events* wird durch das Komponentenmodell derzeit nicht unterstützt.

In der graphischen UML2-Notation werden Assembly Konnektoren nicht immer gezeichnet (siehe etwa Abbildung A.3), die *Requires*-Schnittstelle wird direkt an einer *Provides*-Schnittstelle gezeichnet. Die graphische UML2-Notation nimmt Assembly Konnektoren implizit an. Wird eine solche Abbildung in eine Instanz des Komponenten-Meta-Modells überführt, muss der in der Darstellung abstrahierte Assembly Konnektor dennoch erzeugt werden.

2.7. Zusammengesetzte Komponenten

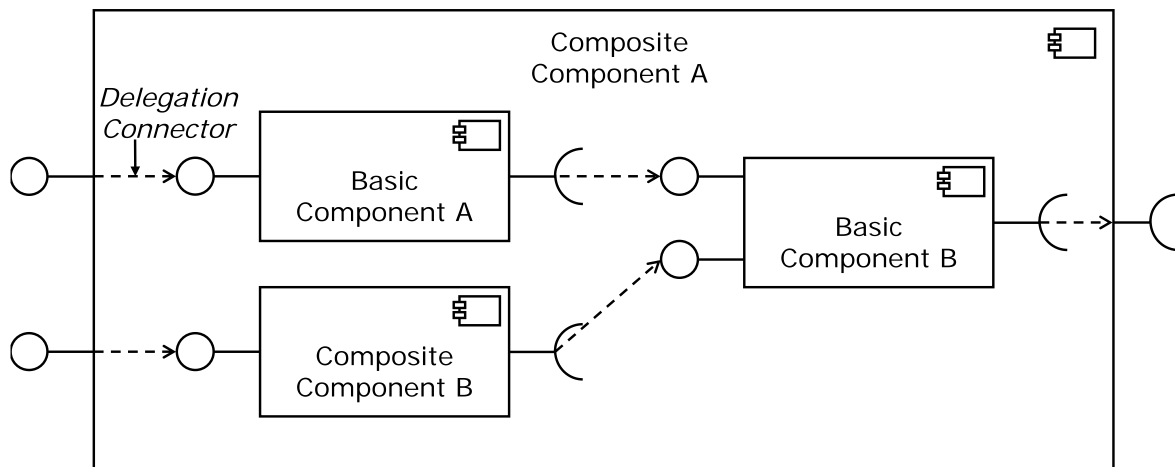


Abbildung 2.5.: Eine zusammengesetzte Komponente mit inneren Komponenten

Zusammengesetzte Komponenten, im Folgenden auch *Composite Components* genannt, enthalten eine interne Realisierung über weitere Komponenten und über zugehörige Strukturen (vgl. Abbildung 2.5). Eine *Composite Component* selbst stellt keine Realisierung über Quellcode dar, sondern realisiert ihr eigenes Verhalten durch die Verwendung von inneren Komponenten (siehe unten). Damit fasst sie eine logisch und / oder funktional zusammengehörige Menge von Komponenten zu einem neuen Komponenten-Typ zusammen.

Über den Mechanismus der *Composite Component* lassen sich Hierarchien von Komponenten definieren. Zudem stellt eine Composite Component eine Abstraktion und Kapselung ihres Innenlebens (*Information Hiding*) zur Verfügung, wenn sie nur über die äußeren Schnittstellen (als „Komponenten-Typ“) betrachtet wird. In Kapitel 2.10 werden die Abstraktionsebenen von Komponenten detailliert aufgezeigt.

Basic Component Verfügt eine Komponente über keine weiteren inneren Komponenten, so wird diese *Basic Component* genannt. Damit bildet sie das Gegenstück zu einer *Composite Component. Basic Components* enthalten Informationen über ihre interne Realisierung in Form von *Service Effect Spezifikationen.* Siehe hierzu Kapitel 2.9.

Composite Components hingegen enthalten Komponenten (*Basic Components* und *Composite Components*). Ihre Realisierung ergibt sich aus den enthaltenen Komponenten. Damit werden tiefere Hierarchiestufen möglich. Im Beispiel (Abb. 2.5) ist „Composite Component B“ eine weitere zusammengesetzte Komponente. Wie zu sehen ist, können *Composite Components* genau wie andere Komponenten verwendet werden. Alle enthaltenen Komponenten einer *Composite Component* sind Teil der „contains“-Relation der zusammengesetzten Komponente.

Delegations-Konnektor In Abbildung 2.5 wird eine exemplarische *Composite Component* dargestellt. Wie zu sehen ist, kann eine *Composite Component* intern eine beliebige Menge von Komponenten aufnehmen. Damit diese Komponenten verwendet werden können, bedient man sich zwischen den Komponenten auf der gleichen Hierarchiestufe der bereits eingeführten Assembly Konnektoren. Zusätzlich reglementieren

Delegations-Konnektoren (*Delegation Connector*), wie auf der Provides-Seite externe Aufrufe auf interne Aufrufe delegiert werden und umgekehrt wie auf der Requires Seite Aufrufe interner Komponente nach außen aus die *Composite Component* heraus delegiert werden.

Auch im Bezug auf die *Composite Components* und Delegations-Konnektoren ist der Kontext bei genauer Betrachtung zu berücksichtigen. Während eine Composite Component *nach außen* über die von ihr angebotenen und benötigten Rollen als Komponenten-Typ aufgefasst werden kann, findet intern eine Verwendung von Komponenten statt. Entsprechend muss der Verwendungskontext von Rollen und Komponenten erfasst werden.

Damit die Abbildung zwischen der äußeren und inneren Rolle einer *Composite Component* eindeutig bleibt, muss stets eine `1:1` Beziehung zwischen äußerer und innerer Rolle bestehen. Ein Delegations-Konnektor verbindet also immer genau zwei Kontext-Rollen (eine äußere angebotene Kontext-Rolle mit einer inneren angebotenen Kontext-Rolle *oder* eine äußere benötigte Kontext-Rolle mit einer inneren benötigten Kontext-Rolle).

Manchmal werden Delegations-Konnektoren zusätzlich über ihren Verwendungsort unterschieden. Findet die Verwendung zum Zwecke einer Abbildung zwischen angebotenen Rollen statt, spricht man auch von *Provides Delegation Connectors*. Hier werden somit genau eine interne Kontext-*Provides Role* und genau eine externe Kontext-*Provides Role* einer *Composite Component* miteinander verbunden. Im umgekehrten Fall der Verwendung zwischen benötigten Schnittstellen wird der entsprechende Delegations-Konnektor auch *Requires Delegation Connector* genannt. Dies entspricht der Verbindung genau einer internen Kontext-*Requires Role* mit genau einer externen Kontext-*Requires Role* einer *Composite Component*.

Ein Signatur-*Mapping* im Sinne eines Adapters (vgl. auch [34, 21]) ist für die Delegations-Konnektoren nicht möglich. In diesem Falle wäre eine Adapterkomponente vorzusehen, die die notwendige Adaption vornimmt.

In der graphischen Visualisierung mit UML erscheinen Delegations-Konnektoren als gestrichelte Verbindungslinie unter Angabe der Delegationsrichtung über einen Pfeil. Die Linie verbindet eine äußere mit einer inneren Schnittstelle. Wird im Sprachgebrauch davon gesprochen, dass ein Delegations-Konnektor zwei Schnittstellen miteinander verbindet, sind, präzise betrachtet, die in der UML-Darstellung mit den Schnittstellen assoziierten Kontext-Rollen gemeint.

Definition: Composite Component Eine *Composite Component* ist zunächst einmal eine einfache Komponente mit angebotenen und benötigten Rollen. Zusätzlich gehören zu einer *Composite Component* die inneren Kontext-Komponenten (`0..*`) und die inneren Konnektoren (`0..*`) (*Assembly Konnektor*, *Provided Delegation Connector*, *Required Delegation Connector*). Da Konnektoren (sowohl Delegation als auch Assembly) auf Kontext-Rollen definiert sind, sind auch Konnektoren vom Kontext abhängig. Interne Schnittstellen werden indirekt über die Navigation von internen Konnektoren zu Rollen erfasst. Nicht zu einer *Composite Component* gehören externe Assembly Konnektoren.

Weitere Anmerkungen zu *Composite Components* finden sich in Kapitel 2.23.1.

2.8. Schnittstellen, Signaturlisten und Protokolle

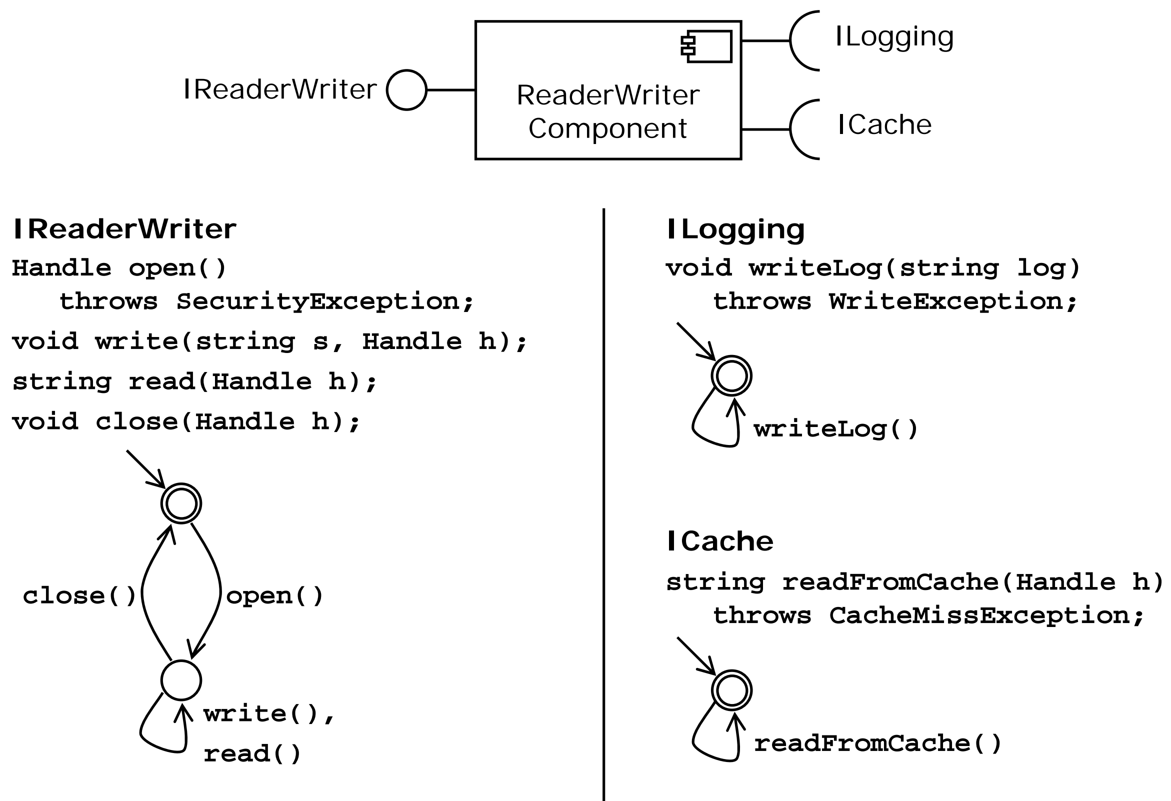


Abbildung 2.6.: Beispielschnittstellen mit Signaturlisten und Protokollinformationen in Form von endlichen Automaten

Bisher wurden Schnittstellen nicht näher charakterisiert. Daher konnten auch keine Interoperabilitätsbedingungen für Komponenten definiert werden. Wesentliche Eigenschaften einer Schnittstelle werden durch die Signaturliste und durch das Protokoll bestimmt. Auf Schnittstellen können innerhalb von Signaturlisten Dienste definiert sein, sowie ein Protokoll, das gültige Aufrufreihenfolgen von Diensten definiert.

Abbildung 2.6 zeigt eine Beispielkomponente „ReaderWriter Component", die die Schnittstelle „IReaderWriter" anbietet und die Schnittstellen „ILogging" und „ICache" benötigt. Diese Komponente wird auch in Kapitel 2.9 als Beispiel verwendet. Zudem wird zu jeder Schnittstelle eine beispielhafte Signaturliste mit dazugehörigem Protokoll im Form eines endlichen Automaten gezeigt.

Zu den Implikationen von Objektzuständen, die sich in Schnittstellen-Protokollen widerspiegeln, siehe auch Nielstraß [57].

2.8.1. Signaturlisten und Signaturen

Jeder Dienst weist auf einer Schnittstelle eine eindeutige Signatur auf, etwa `void doSomething(int a)`. In Anlehnung an Methoden-Signaturen aus Programmiersprachen wie C# und Java sowie der OMG IDL [64], S. 3-1ff haben Signaturen

- einen Rückgabe-Datentyp oder `void` (kein Rückgabewert),

- einen Bezeichner, üblicherweise mit einem sprechenden Namen,
- eine geordnete Menge von Parametern (`0..*`), die jeweils aus einem Datentyp und einem Bezeichner (innerhalb der Parameter eindeutig) bestehen. Zusätzlich können die Modifizierer („Parameter Attribute“) `in`, `out` und `inout` entsprechend der OMG IDL Semantik [64], Kapitel 3 für Parameter verwendet werden.

  In der Standard-Text-Notation für Signaturen werden Parameter in Klammern angegeben und durch Kommata getrennt, Modifizierer werden den Parametern vorangestellt.
- Daneben müssen alle *Exceptions* (Ausnahmen) angegeben werden, die von den Diensten, die durch eine Signatur beschrieben werden, geworfen werden können. Die verpflichtende Angabe von *Exceptions* folgt den Vorgaben von Java. Die Menge der *Exceptions* ist ungeordnet.

  In der Standard-Text-Notation werden *Exceptions* mit dem Schlüsselwort `throws` an die Methoden-Signatur angehängt und durch Kommata getrennt.

Eine Signatur muss für eine Schnittstelle eindeutig über

- Bezeichner und
- Parameter (mit Datentyp und Bezeichner) unter Berücksichtigung der Reihenfolge

bestimmt sein.

Eine Schnittstelle definiert in einer Liste `1..*` Signaturen. Eine Signatur ist genau einer Schnittstelle zugeordnet. Gleichwohl können *verschiedene* Schnittstellen identische Signaturen definieren. Siehe hierzu auch Kapitel 2.23.2.

2.8.2. Protokolle

Wie bereits oben angedeutet, definieren Protokolle von Schnittstellen gültige Aufrufsequenzen auf Diensten. In Abbildung 2.6 werden beispielhaft endliche Automaten zur Darstellung der Protokolle auf den Schnittstellen verwendet. Das Komponentenmodell limitiert dabei Protokolle nicht auf einen bestimmten Typ, wie endliche Automaten oder Petri-Netze, sondern lässt diese Entscheidung bewusst offen. Schnittstellen müssen nicht zwangsläufig Protokolle definieren.

Um im Beispiel zu bleiben: In Abbildung 2.6 beginnen alle gültigen Aufrufsequenzen mit `open()`, gefolgt von Aufrufen von `read()` und `write()` in beliebiger Reihenfolge. In jedem Fall muss abschließend der Aufruf von `close()` folgen. Alternativ sind auch Sequenzen gültig, die sofort enden (denn direkt nach dem Initialisieren befindet sich das Protokoll der Schnittstelle in einem gültigen Endzustand). Als Kantenbeschriftungen werden also Signaturen aus der Signaturliste der mit einem Protokoll zu versehenden Schnittstelle verwendet.

Protokolle bieten somit eine Möglichkeit Abhängigkeiten zwischen Diensten, respektive Signaturaufrufen, *einer* Schnittstelle zu definieren. Damit ist es für Komponenten möglich, einen Teil des internen Zustands in das Komponentenmodell abzubilden. Über Dienstaufrufe kann der Zustand verändert werden. Abhängig vom aktuellen Zustand sind nur bestimmte weitere Dienstaufrufe möglich. Nur bestimmte Sequenzen, nämlich

jene, die zu einem Endzustand führen, sind gültig. Von Komponenten kann nur dann ein definiertes Verhalten erwartet werden, wenn die Schnittstellenprotokolle erfüllt werden.

Zu beachten ist, dass Schnittstellen ein Protokoll definieren können, der Protokollzustand jedoch von der implementierenden und anbietenden (*provides*) Komponente abhängt. Schnittstellen sind also zustandslos. Zieht man einen Vergleich zur objekt-orientierten Programmierung, so sind auch dort Schnittstellen zustandslos. Erst Instanzen von Klassen (Objekte), die Schnittstellen implementieren, besitzen einen Zustand. Analog sind aus komponentenorientiert Sicht Schnittstellen zustandslos. Ein Zustand ist erst bei (Laufzeit-) Instanzen von Komponenten vorhanden.

Einschränkungen Das Komponentenmodell erlaubt indes derzeit keinen Zustandswechsel in Abhängigkeit von Aufrufparametern eines Dienstes. Dennoch gibt es einen möglichen *Work-Around*, der diese Einschränkung in Spezialfällen umgehen kann. Setzt man eine endliche Zahl von Parameterwerten voraus, die für den Aufruf eines Dienstes verwendet werden können, lässt sich ein Verhalten nachbilden, bei dem in Abhängigkeit von Parameterwerten ein Zustandswechsel der Komponente eintritt. Dazu ist es notwendig, die Zahl der Signaturen zu erhöhen. Je Parameterwert, der einen Zustandswechsel bewirken soll, wird eine neue Signatur eingeführt. Anstelle des Aufrufs der gleichen Signatur mit veränderten Parameterwerten, erfolgt ein Aufruf einer anderen Signatur. Eine Komponente kann dann in Abhängigkeit von der aufgerufenen Signatur den internen Zustand wechseln.

Eine Komponente kann mehrere Schnittstellen zugleich anbieten oder benötigen. Als weitere Einschränkung des Komponentenmodells gilt derzeit, dass es innerhalb der Menge der angebotenen *oder* benötigten Schnittstellen keine Möglichkeit gibt, schnittstellenübergreifende Protokolle zu definieren. Eine Komponente hat zu einem Zeitpunkt einen *komplexen* Zustand, der auf mehreren Schnittstellen wahrgenommen werden kann. Je Schnittstellenprotokoll scheint die Komponente einen eigenen Zustand zu haben, der aus dem komplexen Zustand resultiert.

Damit lässt sich jedoch kein sinnvolles Verhalten modellieren, bei dem sich Schnittstellenaufrufe gegenseitig beeinflussen, wie etwa sich gegenseitig blockierende Aufrufe auf Grund einer gemeinsamen kritischen Ressource. Zugriffe auf unterschiedliche Schnittstellen müssen stets unabhängig voneinander durchführbar sein, da sich bspw. das oben genannte blockierende Verhalten nicht erkennen lässt. Siehe hierzu auch Kapitel 2.21.1.

2.9. Service Effekt Spezifikation

Während Protokolle gültige Aufrufsequenzen auf Schnittstellen beschreiben, stellen Service Effekt Spezifikationen, auch SEFF (*Service Effect Specification*; abstrakt betrachtet), eine Verbindung zwischen angebotenen und benötigten Schnittstellen einer Komponente her. SEFFs sind grundsätzlich für *Composite Components* und *Basic Components* möglich. Die SEFFs von *Composite Components* sollten jedoch aus den SEFFs innerer Komponenten berechnet werden, wohingegen *Basic Components* eine Realisierung von SEFFs spezifizieren können, die nicht auf weitere innere Komponenten verweist. Daher limitiert das Komponentenmodell die Möglichkeit SEFFs zu spezifizieren auf *Basic Components*. Die SEFFs von *Composite Components* werden

entsprechend des in Kapitel A.2 skizzierten Verfahrens (für endliche Automaten) berechnet.

Wie der Name andeutet, beschreibt ein SEFF zu genau einem angebotenem Dienst (*Service* des *Provided Interfaces*) einer Komponente die externen Auswirkungen auf die benötigten Schnittstellen der Komponente. Interne Vorgänge in der Komponente werden dabei abstrahiert. Dafür werden sämtliche Aufrufe auf externen Komponenten (über die benötigten Schnittstellen) erfasst. Damit ist es möglich die Auswirkungen eines Dienstaufrufs auf einer angebotenen Schnittstelle einer Komponente auf den benötigten Schnittstellen der Komponenten zu beobachten.

Der SEFF zu einer Komponente erfasst zusätzlich Kontrollfluss-Elemente, sofern diese Kontrollfluss-Elemente Auswirkungen auf die externen Aufrufe haben. So werden beispielsweise Kontrollfluss-Verzweigungen über `if`-Abfragen in einen SEFF aufgenommen, sofern in beiden Ästen des verzweigten Kontrollflusses externe Dienstaufrufe erfolgen.

Für *Basic Components* und *Composite Components* muss ein SEFF verpflichtend definiert werden. Im Falle von *Composite Components* ist der SEFF, wie in Kapitel A.2.2 dargestellt, berechnet.

ReaderWriter Component

```
ILogging logger;
ICache cache;
[..]
public string read(Handle h) {
   logger.writeLog("start cache read");
   while(h.hasNext()) {
        cache.readFromCache(h.current());
        logger.writeLog("cache access");
   }
   logger.writeLog("end cache read");
}
```

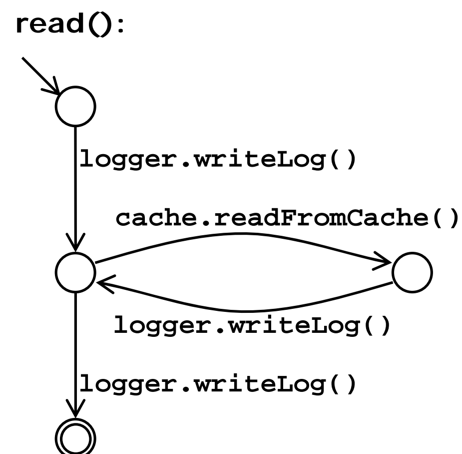


Abbildung 2.7.: Service Effekt Spezifikation als endlicher Automat zu beispielhaftem Pseudo-Quellcode

In Abbildung 2.7 wird eine Service Effekt Spezifikation zu der in Abbildung 2.6 eingeführten Komponente dargestellt. In diesem Beispiel wird wiederum auf endliche Automaten (*Finite State Machine*, FSM) zur Darstellung von SEFFs zurückgegriffen. Auch an dieser Stelle beschränkt das Komponentenmodell nicht auf die Verwendung von endlichen Automaten. Ebenso sind Petri-Netze, reguläre Ausdrücke[1] oder andere Sprachen beschreibenden Konstrukte möglich.

Die möglichen Typen von SEFFs müssen in der Lage sein, den abstrakten SEFF abzubilden. Dabei muss eine bijektive Abbildung zwischen dem abstrakten SEFF und dem konkret gewählten SEFF-Typen möglich sein. Diese bijektive Abbildung betrifft:

[1] Für die Zukunft wird angestrebt, reguläre Ausdrücke als Standard für SEFFs (und auch Protokolle) zu verwenden.

- die Benennung des angebotenen Dienstes der Komponente, zu der der SEFF angegeben werden soll,
- eine Benennung der Dienste, die auf benötigten Schnittstellen aufgerufen werden
- die Festlegung möglicher Folgen von Aufrufen auf den benötigten Diensten.

Da die Verwendungsreihenfolge von geforderten Diensten auch vom Zustand einer Komponente oder den Aufrufparametern eines angebotenen Dienstes abhängen kann, werden *SEFFs* beispielsweise als endliche Automaten dargestellt, damit auch Bedingungen (beispielsweise `if`-Ausdrücke) oder Schleifen (etwa `while`) erfasst werden können.

Im Beispiel werden alle Initialisierungsvorgänge der Komponente (Konstruktoraufrufe, Variablen anlegen) ausgeblendet. Der erste Aufruf, der von Interesse ist, ist der Aufruf des Loggers (`logger.writeLog()`). Als nächstes werden die Kontrollfluss-Elemente der Komponente abgebildet. Da `cache.readFromCache()` und `logger.writeLog()` in einer `while`-Schleife aufgerufen werden können, wird ein Zwischenzustand eingeführt, der den Zustand nach dem Aufruf des Cache-Zugriffs repräsentiert und ein Kante, die zum Zustand vor / nach der `while`-Schleife zurück führt.

Bei SEFFs werden die Kanten eines endlichen Automaten mit den externen Dienstaufrufen beschriftet. Die Zustände des Automaten repräsentieren den „Zustandsbereich" der Komponente zwischen den externen Aufrufen.

Implementierungen Interpretiert man die Menge aller gültigen Aufrufsequenzen eines SEFFs als Wörter einer Sprache, muss die Menge der Wörter in dieser Sprache größer oder gleich der real von einer Implementierung einer Komponente akzeptierten Menge von Wörtern sein. Ein SEFF muss also eine Obermenge ($SEFF - Sprache \supseteq Implementierungssprache$) aller gültigen Aufrufe auf der benötigten Schnittstelle darstellen. Einer Implementierung einer Komponente bleibt es damit freigestellt, ob sie beispielsweise die Möglichkeit nutzt, eine Schleife unendlich oft zu durchlaufen oder nur eingeschränkt oft.

Es sei angemerkt, dass in der Semantik von *Provided Component Types*, siehe. Kapitel 2.10, SEFFs keinen verpflichtenden Charakter haben.

SEFF-Typen Für jeden Dienst einer Komponente können `0..1` SEFFs eines SEFF-Typs definiert werden. Als SEFF-Typ gelten dabei endliche Automaten, Petri-Netze und sonstige „sprachbeschreibende Konstrukte". Existieren zu einem Dienst einer Komponente SEFF-Beschreibungen unterschiedlichen Typs, so müssen diese konsistent sein, dürfen sich also nicht widersprechen. Das Komponentenmodell prüft zunächst in keinem Fall auf diese Konsistenz. Entsprechende Prüfungen müssen durch den Nutzer erfolgen. Eine Prüfung ist nicht in jedem Fall trivial, da beispielsweise die Sprachinklusion zwischen den von unterschiedlichen SEFF-Typen akzeptierten Sprachen erfolgen muss.

Das Komponentenmodell verzichtet bewusst auf die Einschränkung, nur einen SEFF-Typ global im gleichen Komponentenmodell oder für einen Dienst zuzulassen. Dies erlaubt es, die gleiche Komponentenarchitektur zur gleichen Zeit über verschiedene SEFF-Typen zu spezifizieren. Dadurch können Modellierer, auch ohne Kenntnis von der spezifischen Darstellung von SEFFs, ihre eigenen Darstellungsformen verwenden. Im

Sinne der Unabhängigkeit des Komponentenmodells von konkreten SEFF-Typen bleibt es durch dieses Weniger an Einschränkung möglich, das Komponentenmodell gefiltert mit einer Auswahl von SEFF-Typen zu betrachten. Zugleich erkauft man sich diesen Vorteil durch die Gefahr, dass Inkonsistenzen verschiedener SEFF-Typen auftreten oder spezielle SEFF-Typen schlicht nicht verstanden werden und dadurch auch nicht konsistent zu halten sind.

Einschränkungen Je nach der Wahl eines konkreten SEFF-Typen sind damit unterschiedliche Einschränkungen verbunden. So sollte bedacht werden, dass beispielsweise endliche Automaten im Gegensatz zu Petri-Netzen keine parallelen Abläufe und anschließende Synchronisation oder eine endliche Begrenzung von Schleifen-Durchläufen unterstützen. Die Mächtigkeit des Komponentenmodells hängt daher im Bereich der SEFFs erheblich von der Wahl des konkreten SEFF-Typen ab.

2.9.1. Parametrisierte Verträge

Parametrisierte Verträge, wie sie von Reussner in [71] beschrieben werden, können im Komponentenmodell über SEFFs spezifiziert werden und ermöglichen die Definition von Abhängigkeiten zwischen *Provides*- und *Requires*-Schnittstelle.

Im Kern kann die Idee von parametrisierten Verträgen wie folgt skizziert werden: Wird eine *Requires*-Schnittstelle lediglich begrenzt bedient – die mit der *Requires*-Schnittstelle konnektierte *Provides*-Schnittstelle bietet also nicht den vollen Umfang – so kann die *Provides*-Schnittstelle unter Umständen dennoch begrenzt erfüllt werden. Ein parametrisierter Vertrag definiert die Abhängigkeit der Dienste, die von einer Komponenten erbracht werden, von benötigten Diesten der gleichen Komponente und damit exakt, mit welchen Einschränkungen die *Provides*-Schnittstelle, respektive der dahinter stehende Vertrag, bei einer konkreten Erfüllung der *Requires*-Schnittstelle angeboten werden kann.

Sollen zusätzliche Eigenschaften von Verträgen von Komponenten angegeben werden, kann dabei auf Annotationen (vgl. Kapitel 2.19) zurückgegriffen werden. Eine Auswertungsfunktion für parametrisierte Verträge ist nicht Teil des Palladio Komponentenmodells.

2.9.2. Quality of Service

Nachdem vorgestellt wurde, dass Komponenten über Protokolle und SEFFs mit zusätzlichen Informationen angereichert werden können, lassen sich damit Berechnungen und Analysen vornehmen. Zur Bestimmung von Quality of Service-Daten wird vor allem auf die Informationen von SEFFs zugegriffen.

Quality of Service-Daten lassen sich über Annotationen erfassen. Dabei lassen sich QoS-Daten aus Messungen hinterlegen, berechnete Werte zu Entitäten speichern und aus beiden Formen von Werten weitere Berechnungen vornehmen. Auch SEFFs sind mit QoS-Daten annotierbar, weshalb sich über komplette Komponentenarchitekturen hinweg Berechnungen vornehmen lassen.

Da eine beliebige Zahl von Annotationen zur gleichen Zeit für eine Entität denkbar sind, sind parallele Berechnungen verschiedener QoS-Aspekte an der gleichen Modellinstanz, ohne, dass sich unerwünschte Seiteneffekte ergeben, möglich. Damit ist ein

und dieselbe Modellinstanz zur gleichen Zeit aus verschiedenen Sichten examinierbar.

2.10. Komponenten: Typ-Ebenen

Das Palladio Komponentenmodell umfasst das Konzept verschiedener Modellierungs-Ebenen für eine Modell-Ausprägung. Die Modellierungs-Ebenen spiegeln sich dabei in Ebenen eines sich detaillierenden Typ-Begriffs für Komponenten wider. Dies bedeutet, dass es möglich ist, auf verschiedenen Abstraktionsniveaus eine Komponentenarchitektur zu modellieren.

Damit wird ein *Top-Down*-Vorgehen (vgl. [25], S. 39) bei der Modellierung unterstützt. Es ist möglich, gezielt Komponentendetails hinzuzufügen. Dabei werden für die Detaillierung durch das Komponentenmodell *Conforms*-Beziehungen definiert, denen die Verfeinerung folgen muss. Somit lassen sich Verfeinerungen konsistent über Typ-Ebenen hinweg durchführen.

In der anderen Richtung findet sich die Möglichkeit zur *Bottom-Up*-Modellierung im Komponentenmodell. Durch die Verwendung existierender Komponenten, lassen sich detaillierte Informationen zu Komponenten-Typen gewinnen, die dann schrittweise abstrahiert werden können. Über die Typ-Ebenen wird für diesen Fall reglementiert, welche Informationen weggelassen werden müssen. Detailbeschreibungen existierender Komponenten lassen sich beispielsweise über *Reflection*-Mechanismen (vgl. [47], S. 412ff für .NET) gewinnen.

Da das Komponentenmodell zur gleichen Zeit Komponenten verschiedener Typ-Ebenen – im Folgenden auch Typ-Niveaus genannt – in einem Modell unterstützt, sind gemischte Vorgehen aus *Top-Down* und *Bottom-Up* „Gegenstrom“ möglich. Detailinformationen können somit auch nur jenen Komponenten hinzugefügt werden,

- die aktuell von Interesse sind,
- die bei iterativem Vorgehen bereits behandelt wurden,
- über die überhaupt Detailinformationen vorliegen,
- die die Informations-Komplexität nicht sprengen und unter Umständen die Berechenbarkeit einer Modellinstanz erhalten.

Komponenten-Typen stellen eine Außensicht auf Komponenten dar. Sie definieren nach außen hin attestierbare Anforderungen an Komponenten.

2.10.1. Typ-Hierarchie

Im Komponentenmodell wird in drei Typ-Ebenen für Komponenten unterschieden, die einander in der angegebenen Reihenfolge nach unten spezialisieren:

- *Provided Type*
- *Complete Type*
- *Implementation Type*

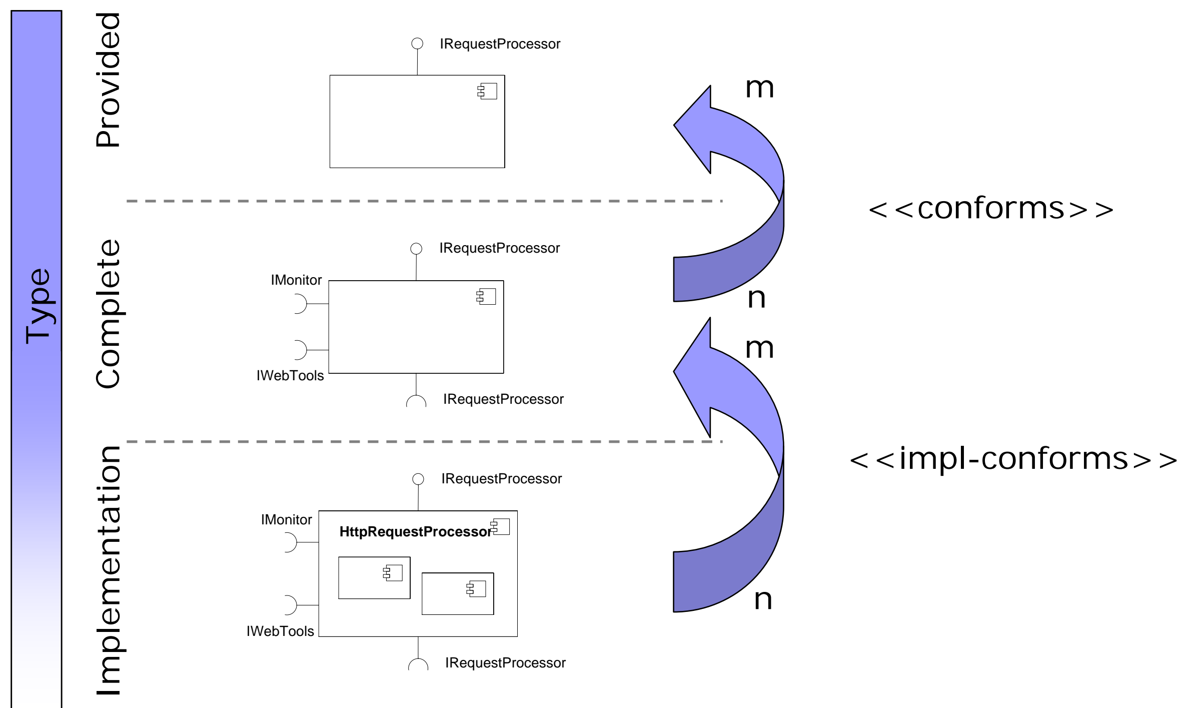


Abbildung 2.8.: Typ-Hierarchie: *Provided Type*, *Complete Type* und *Implementation Type* (nach [10], S. 18)

In Abbildung 2.8 wird die Typ-Hierarchie des Komponentenmodells dargestellt. Zwischen den Typ-Ebenen besteht eine *Conforms*-Beziehung, wobei sich diese zwischen (*Provided Type* und *Complete Type*) und (*Complete Type* und *Implementation Type*) unterscheidet. Zur *Conforms*-Beziehung siehe auch Kapitel 2.11.

Die farbliche Hinterlegung des „Typ-Balkens" auf der linken Seite deutet an, dass der *Provided Type* und *Complete Type* dem „klassischen" Typ-Verständnis (vgl. [7, 6]) entsprechen, indem die angebotenen Schnittstellen definiert werden, wohingegen der *Implementation Type* auch Vorgaben für die interne Realisierung eines Typs macht. Letzteres wird klassischerweise weniger als Typ angesehen.

Würde man die Typ-Hierarchie nach unten weiter führen, würde sich unter anderem ein Laufzeit-Typ anschließen, der die Laufzeiteigenschaften einer Komponente reglementiert. Diese Typ-Ebene wird im Komponentenmodell jedoch nicht betrachtet. Über den bloßen Komponenten-Typ hinaus gehen die in Kapitel 2.16 betrachteten weiteren Ebenen des Komponentenmodells.

2.10.2. Komponenten-Typen

Provided Type In Abbildung 2.8 wird der *Provided Type* als oberste Ebene dargestellt.

Ein *Provided Type* ist ein Komponenten-Typ, der seine angebotenen Schnittstellen verbindlich definiert. Zusätzlich darf er benötigte Schnittstellen angeben, jedoch handelt es sich hierbei um eine unverbindliche Angaben. Die von einem *Provided Type* angegebenen benötigten Schnittstellen können beispielsweise zur Speicherung von Modellierungsideen genutzt werden. Damit haben die als benötigt angegebenen Schnittstellen keine weitere Semantik.

Complete Type In Abbildung 2.8 wird der *Complete Type* als mittlere Ebene dargestellt.

Ein *Complete Type* definiert seine angebotenen und benötigten Schnittstellen verbindlich. Dabei muss er *alle* benötigten Schnittstellen angeben.

Implementation Type In Abbildung 2.8 wird der *Implementation Type* als unterste Ebene dargestellt.

Neben den Anforderungen, die ein *Complete Type* erfüllen muss, muss der *Implementation Type* die interne Realisierung der Komponente wie folgt festlegen.

- Ist der *Implementation Type* als *Composite Component* realisiert, bedeutet das, dass alle internen Komponenten, die direkt in einer derart typisierten Komponente enthalten sind, mit samt der assoziierten Assembly Konnektoren und Delegations-Konnektoren definiert werden müssen.

- Erfolgt die Realisierung als *Basic Component*, können keine inneren Verbindungsstrukturen analog zu einer *Composite Component* spezifiziert werden. Dafür müssen SEFFs zu allen angebotenen Dienste des *Implementation Types* spezifiziert werden.

Bemerkungen *Composite Components* können zur gleichen Zeit Kontext-Komponenten von Komponenten-Typen beliebiger Typ-Ebenen in ihrem Inneren vereinen, sich also intern aus unterschiedlich detaillierten Komponenten zusammensetzen.

Sowohl *Composite Components* als auch *Basic Components* sind *Implementation Types*. Sie erben direkt von der Meta-Klasse „Implementation Type“, wie in Abbildung 2.9 dargestellt wird. Ein *Implementation Type* muss entweder von einer *Composite Component* oder einer *Basic Component* realisiert werden («`realizes`»-Stereotyp).

2.10.3. Darstellung und Ebenenbeziehungen

2.10.3.1. Vorbemerkung

In Abbildung 2.9 werden neben der Vererbungsstruktur der Komponenten-Typen untereinander die *conforms*-Beziehungen zwischen den Komponenten-Typ-Ebenen aufgezeigt. Die *conforms*-Beziehung zwischen zwei Komponenten-Typen wird als Assoziation dargestellt. Hierbei handelt es sich jedoch lediglich um ein Hilfskonstrukt. Ein Komponenten-Typ muss den Komponenten-Typ, zu dem er in einer *conforms*-Beziehung steht, nicht explizit referenzieren. Vielmehr stehen zwei Komponenten-Typen automatisch untereinander in einer *conforms*-Beziehung, sobald sie die im Folgenden genannten Bedingungen erfüllen.

Eine Diskussion der Modellierung der Vererbungsstruktur und der *conforms*-Beziehungen erfolgt in Kapitel 3, das sich mit der Modellierung auseinandersetzt. In diesem Kapitel werden dagegen die Semantik der *conforms*-Beziehungen und die Grundideen der Komponenten-Typen dargelegt.

Im Folgenden werden die Begriffe Sub-Typ und Implementierung verwendet. Eine Klärung der Begriffe wird im Glossar in Kapitel A.6 vorgenommen.

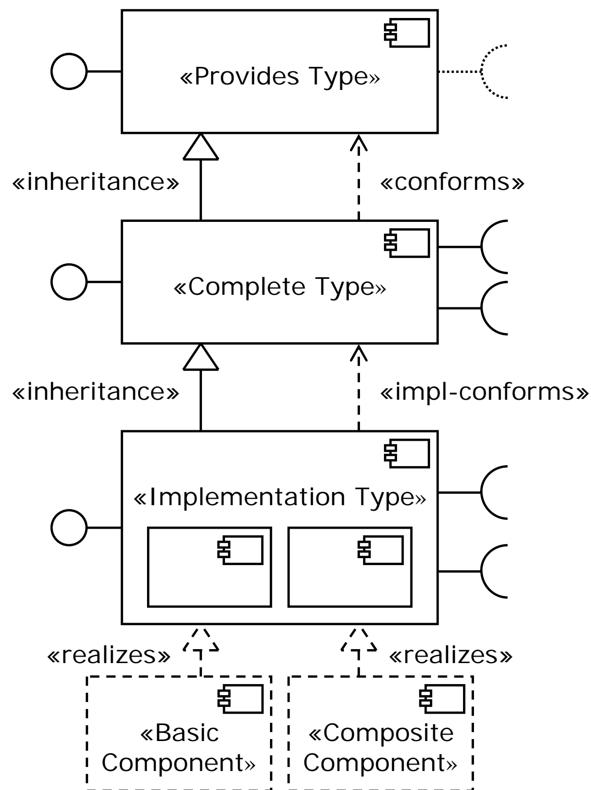


Abbildung 2.9.: Komponenten Typ-Ebenen: Vererbungs- und Konformitätsbeziehungen

2.10.3.2. Darstellung

Bei der Darstellung von Komponenten-Typen sollte, wie in Abbildung 2.9 angegeben, die Typ-Ebene als Stereotyp angegeben werden. Angedeutet durch gestrichelte Linien wird, dass *Provides Types* benötigte Schnittstellen nicht verbindlich angeben. *Implementation Types* geben in ihrem Innern ihre Realisierung an. Anders als in der Abbildung, werden die internen Konnektoren und Schnittstellen dabei angegeben.

2.10.3.3. Conforms-Beziehung

Wie Abbildung 2.9 aufzeigt, bestehen zwischen den Typ-Ebenen der Komponenten des Komponentenmodells paarweise jeweils zwei Beziehungen: Zum einen erben die Komponenten-Typen jeweils von ihrem Vater-Typ – sofern vorhanden – zum anderen können Komponenten jeweils untereinander in einer Konformitätsbeziehung (*conforms* / *impl-conforms*) stehen. Das erste bedeutet, dass die Eigenschaften und der Informationsgehalt der Komponententypen sich ebenfalls nach unten vererben. Weiter unten stehende Komponenten-Typen enthalten also immer mehr Informationen.

Über die *conforms*-Beziehungen wird festgelegt, welchen Regeln Ausprägungen von Komponenten-Typen folgen müssen, damit sie *Sub-Typen* von einander sind. Stehen also zwei Komponenten-Typen zu einander in *conforms- impl-conforms*-Beziehung, folgt daraus, dass sie in einer Sub-Typ- / Super-Typ-Beziehung unter einander stehen.

Somit bezieht sich die Konformität auf konkrete Ausprägungen von Komponenten untereinander, wohingegen die Vererbung die Eigenschaften der Komponenten-Typen definiert.

Das bedeutet weiter, dass zum Beispiel jeder *Complete Type* einen *Provided Type* hat, für den die «conforms» Beziehung erfüllt ist, nämlich den, den er durch die Vererbungsbeziehung selbst beschreibt. So ist jeder *Implementation Type* gleichzeitig ein *Complete Type* und ein *Provides Type*. Damit gilt ebenfalls, das jeder *Complete Type* zu sich selbst in «conforms»-Beziehung steht und ein *Implementation Type* zu sich selbst in «impl-conforms»- und «conforms»-Beziehung steht.

Conforms-Beziehung zwischen Complete Type und Provided Type Ein *Complete Type* steht zu einem *Provided Type* in *Conforms*-Beziehung, wenn der *Complete Type* auf der *Provides*-Seite die gleichen Schnittstellen anbietet wie der *Provided Type* oder Schnittstellen anbietet, die zu den ursprünglich angebotenen Schnittstellen in Sub-Typ-Beziehung steht. Zudem darf ein *Complete Type* zusätzliche Schnittstellen gegenüber dem *Provided Type* anbieten.

Für die *Requires*-Seite des *Complete Types* gelten keine Vorgaben, da die Definition von benötigten Schnittstellen durch den *Provided Type* unverbindlich erfolgt.

Impl-Conforms-Beziehung zwischen Implementation Type und Complete Type Soll ein *Implementation Type* zu einem *Complete Type* in *Impl-Conforms*-Beziehung stehen, darf der *Implementation Type* angebotene Schnittstellen hinzufügen und weniger Schnittstellen benötigen. Ein gültiger Sub-Typ muss dabei den Vorgaben von Co- und Contra-Varianz folgen, wie sie in Kapitel 2.11.2 vorgestellt werden.

In Analogie zur *Conforms*-Beziehung zwischen *Provided Type* und *Complete Type* darf auch zwischen *Complete Type* und *Implementation Type* die Co- und Contra-Varianz *auf* Schnittstellen genutzt werden. Auf der angebotenen Seite dürfen also mehr Dienste auf einer Schnittstelle definiert werden, auf der benötigten Schnittstelle weniger.

2.10.3.4. Charakterisierung von Komponenten-Typen

Provided Type Ein *Provided Type* enthält keine Informationen über die interne Realisierung, darf diese Informationen also nicht spezifizieren. Dies bedeutet, dass alle Sub-Typen eines konkreten *Provided Type* in der internen Realisierung frei sind und ebenfalls keine Vorgaben über die benötigten Schnittstellen einhalten müssen. Damit ist der *Provided Type* der Typ, der für Sub-Typen am wenigsten Anforderungen definiert. Auf diese Weise werden Komponenten definiert, deren Verwendungsmöglichkeiten im Vordergrund stehen.

Complete Type Bietet ein *Complete Type* den Sub-Typ einer Schnittstelle an, die von einem *Provided Type* angeboten wird, so ist der *Complete Type* damit in der Lage unter Ausnutzung der Sub-Typ-Beziehungen auf Schnittstellen mehr Dienste anzubieten als der *Provided Type*, zu dem eine *Conforms*-Beziehung besteht. Siehe hierzu auch Kapitel 2.11.

Wie auch der *Provided Type* darf der *Complete Type* keine Informationen oder Vorgaben über die interne Realisierung enthalten. Daher sind auch hier die Sub-Typen und Implementierungen eines *Complete Types* bezüglich der internen Realisierung frei.

Implementation Type Durch ihre Fähigkeit Vorgaben über ihre Realisierung durch *Basic* und *Composite Components* zu machen, stellen *Implementation Type* den detailliertesten Komponenten-Typ des Komponentenmodells dar.

Bemerkung zu SEFFs Da durch die Pflicht zur Spezifikation von SEFFs für *Basic Components* und *Composite Components* das *Requires Protocol* berechnet werden kann, muss dieses konsistent zu den Realisierungen zwischen den Typ-Ebenen bleiben.

2.10.4. Quantitäten der Sub-Typ-Beziehungen

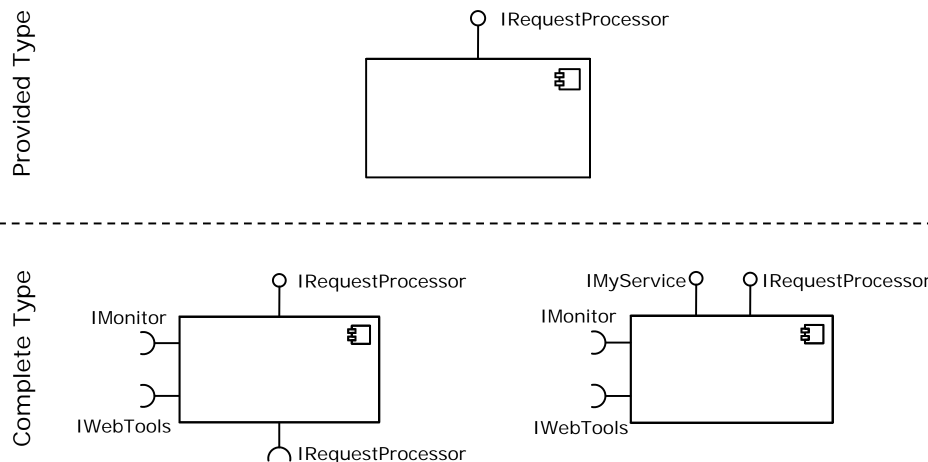


Abbildung 2.10.: Spezialisierung eines *Provided Type* zu zwei *Complete Types* (nach [10], S. 17)

1:n-Relation Zunächst einmal soll anhand von Abbildung 2.10 illustriert werden, dass es zu einem *Provided Type* eine beliebige Menge (`0..*`) von *Complete Types* geben kann. Solange das/die durch den *Provides Type* vorgegebene(n) angebotene(n) Interface(s) (im Beispiel `IRequestProcessor`) oder ein Sub-Typ dieser Schnittstelle vom *Complete Type* angeboten wird/werden, handelt es sich bei den Komponenten um Sub-Typen. Sub-Typen folgen der *conforms*-Beziehung (siehe Abb. 2.8) zwischen den betroffenen Typ-Ebenen für Komponenten. Der *Complete Type* kann so, wie im Beispiel mit `IMyService`, zusätzliche Schnittstellen anbieten. Außerdem sind *Complete Types* komplett bezüglich der benötigten Schnittstellen frei. Im Beispiel wird die Schnittstelle `IRequestProcessor` lediglich von der linken Komponente angeboten. Auch die im Beispiel überschneidend angeführten Schnittstellen `IMonitor` und `IWebTools` sind nicht zwischen zwei Sub-Typen des gleichen *Provided Types* erforderlich.

m:1-Relation Abbildung 2.8 zeigt eine weitere Besonderheit zwischen den Typ-Ebenen auf. Ein *Complete Type* kann der Sub-Typ einer beliebigen Anzahl (0..*) von *Provides Types* sein und ein *Implementation Type* kann Sub-Typ einer beliebigen Anzahl von *Complete Types* sein.

Die `m:1`-Beziehung zwischen *Provides Type* und *Complete Type* ist möglich, da, wie oben beschrieben, ein *Complete Type* ebenfalls Schnittstellen anbieten kann, die von

einem konkreten *Provides Type* nicht spezifiziert wurden. Dafür können diese „zu viel“ angebotenen Schnittstellen von anderen *Provides Types* angeboten werden. Prinzipiell kann somit ein *Complete Type* zugleich Sub-Typ einer Menge von *Provides Types* sein.

Auch die `m:1`-Beziehung zwischen *Complete Type* und *Implementation Type* gründet auf der gleichen Möglichkeit, dass ein *Implementation Type* zugleich Sub-Typ zweier *Complete Types* ist. Bietet ein *Implementation Type* die Schnittstellen „I1“ und „I2“ an, so kann es zwei *Complete Types* geben, von denen einer ebenfalls „I1“ anbietet, der andere jedoch „I2“. Damit gibt es zu einem *Implementation Type* zwei, und im Allgemeinen m, Super-Typen.

m:n-Relation Insgesamt ergeben sich somit zwischen den „benachbarten“ Typ-Ebenen des Komponentenmodells `m:n`-Beziehungen. Da die Sub-Typ-Beziehung transitiv ist, folgt daraus, dass diese Bedingung für alle Ebenen gilt.

2.11. Interoperabilität

Die Interoperabilität von Komponenten definiert, welche Komponenten über welche Schnittstellen miteinander interagieren können. Für den allgemeinen Fall wird Interoperabilität für je eine Schnittstelle zweier Komponenten definiert, die miteinander verbunden sind.

Im folgenden Kapitel wird Interoperabilität von Komponenten anhand des Beispiels der Substitution von Komponenten dargestellt. Dies hat den Vorteil, dass zugleich die *Requires*- und *Provides*-Seite betrachtet werden und eine Interoperabilitätsprüfung für alle angebotenen und benötigten Schnittstellen der zu substituierenden Komponente erfolgt.

2.11.1. Substitution von Komponenten

Nicht immer implementieren „miteinander verbundene“ Komponenten exakt die gleiche Schnittstelle. Wird eine Komponentenarchitektur mit realen Komponenten modelliert, werden zwangsläufig auch bestehende Komponenten erfasst, die fest und unabänderlich (wenn man von Adaption absieht) bestimmte Schnittstellen anbieten und benötigen. Dennoch ist es wünschenswert in bestehenden Komponentenarchitekturen gezielt Komponenten durch andere austauschen zu können.

Daher stellt sich für Assembly Konnektoren die Frage, ob und welche Verbindungen nach einer Substitution gültig sind. Abhängig davon, wie streng der Typ-Begriff gefasst wird, also entsprechend der in Kapitel 2.10.1 definierten Komponenten-Typen, sind die Konditionen, denen eine Ersetzung von Komponenten unterliegt, unterschiedlich definiert.

Abbildung 2.11 verdeutlicht die Problematik, die sich bei der Substitution von Komponenten durch andere ergibt. Die auf der linken Seite gegebene „Component A“ benötigt die Schnittstelle `IAr`. Auf der rechten Seite gibt „Component B“ die Schnittstelle `IBp` fest vor. An dieser Stelle ist zu klären, welche Komponenten (mit welchen angebotenen und benötigten Schnittstellen) für „Component X“ verwendet werden können.

Das skizzierte Problem besteht ebenso für Komponentenarchitekturen, die auf Grundlage existierender Komponenten entwickelt werden sollen. Dann ist zu klären, welche

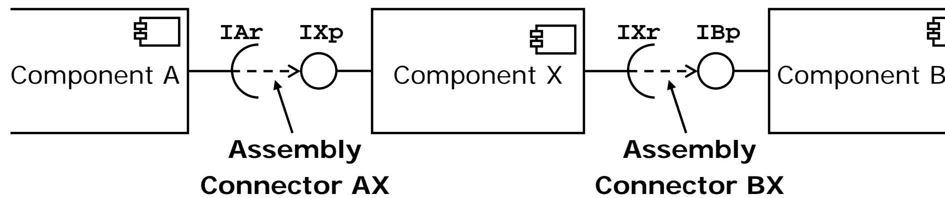


Abbildung 2.11.: Szenario in dem Komponente X („Component X") in einer bestehenden Komponentenarchitektur substituiert werden soll

bestehenden Komponenten mit welchen anderen Komponenten sinnvoll interagieren können.

Zu bedenken bleibt, dass, je nach Detailniveau respektive Typ-Ebene der Komponenten, auf den Schnittstellen Protokolle definiert sein können und die Komponenten selbst SEFFs vorschreiben. Bei einer Komponenten-Substitution sind gegebenenfalls auch diese zu berücksichtigen.

Die genaue Definition gültiger Substitutionen ist variabel und nicht durch das Komponentenmodell festgeschrieben. Kapitel 2.20 beschreibt Möglichkeiten zur dynamischen Definition von Validität. Daher können an dieser Stelle nur allgemeine, dem Komponentenmodell zu Grunde liegende, Regeln dargestellt werden.

2.11.2. Co- und Contra-Varianz

Allgemein Interoperabilität unter Ausnutzung von „Co-Varianz" (siehe auch [1, 49]) für miteinander verbundene Komponenten liegt im Allgemeinen dann vor, wenn auf allen *Provides Interfaces* einer Komponente

- die Menge der angebotenen Dienste eine Obermenge der geforderten Dienste ist, und
- das angebotene Protokoll eine Sprach-Obermenge des geforderten Protokolls ist.

Umgekehrt muss für alle *Requires Interfaces* gelten, dass

- die geforderten Dienste eine Untermenge der angebotenen Dienste sind, und
- das geforderte Protokoll eine Sprach-Untermenge des angebotenen Protokolls ist.

Genügen alle *Provides* und alle *Requires* Schnittstellen einer Komponente den oben genannten Bedingungen, kann sie (als Substitut) verwendet werden.

Genauere Interoperabilitätsbedingungen werden nicht durch das Komponentenmodell selbst, sondern durch externe Algorithmen festgelegt. Hier folgt das Komponentenmodell wiederum der Idee des *Strategy*-Musters und trennt zwischen der Funktion des Komponentenmodells als Datencontainer mit *grundlegenden* Interoperabilitätsbedingungen und der Anwendung eigener unterschiedlicher Strategien zur Bestimmung näherer Interoperabilitätsbedingungen.

Conforms-Beziehungen Die aufgestellten Interoperabilitäts-Bedingungen gelten mit Ausnahme der Forderungen für Protokolle und SEFFs automatisch zwischen Komponenten-Typen, die untereinander in ≪conforms≫ bzw. ≪impl-conforms≫-Beziehungen

stehen. Diese Beziehungen garantieren die Erfüllung von Co- und Contra-Varianz für Komponenten-Typen untereinander. So kann beispielsweise eine Komponente A durch Komponente B ersetzt werden, falls gilt: $B \ll impl-conforms \gg A_{compl}$, wobei A_{compl} der von A definierte *Complete Type* ist.

Protokolle Um entscheiden zu können, ob Komponenten durch andere substituiert werden können, ist es sinnvoll, Protokolle, respektive Sprachen, zu verwenden, deren Inklusionsproblem entscheidbar ist. Wie bereits zuvor erwähnt wurde, ist die Verwendung von endlichen Automaten im Komponentenmodell sehr verbreitet. Da endliche Automaten und reguläre Ausdrücke sprachäquivalent sind, und für reguläre Ausdrücke das Inklusionsproblem lösbar ist, womit diese Problem auch für endliche Automaten lösbar ist, erscheint die Entscheidung zur Verwendung von endlichen Automaten in Praxisbeispielen sinnvoll. Die andere, im Bereich des Komponentenmodells verbreitete Variante zur Darstellung von Protokollen und SEFFs, die Petri-Netze, sind über kontextsensitive Grammatiken darstellbar, weshalb das Wortproblem ebenfalls lösbar ist (vgl. [79]: „Da die Sprachen von Petri-Netzen sämtlich kontextsensitiv ist, ist das Wortproblem entscheidbar.“).

Generell sollte bei der Wahl von Protokoll-Sprachen bedacht werden, dass diese für Interoperabilitätsprüfungen automatisch prüfbar sein sollten.

Trivial-Protokoll Da Protokolle auf Schnittstellen, unabhängig von der Typ-Ebene von Komponenten, definiert werden *können*, gibt es keine Kopplung zwischen vorliegenden Komponenten einer bestimmten Typ-Ebene und dem Vorkommen von Protokollen. Es ist jeweils möglich, dass ein Protokoll definiert ist, eine Sicherheit gibt es hierfür nicht. Um dennoch die Substituierbarkeit unter Beachtung von Protokollen prüfen zu können, nimmt das Komponentenmodell, sofern auf einer Schnittstelle kein Protokoll definiert ist, ein Trivial-Protokoll an. Abbildung 2.12 zeigt ein Trivial-Protokoll. Bei diesem implizit angenommenen Protokoll sind Dienstaufrufe auf der Schnittstelle in beliebiger Reihenfolge möglich.

Soll eine Komponente substituiert werden, die auf einem Teil ihrer Schnittstellen kein Protokoll definiert hat, wird für diese Schnittstellen das Trivial-Protokoll (siehe Abbildung 2.12) zur Prüfung anhand der in Kapitel 2.11.2 genannten Regeln herangezogen. Die Interoperabitätsprüfung ist damit ebenso möglich wie bei Schnittstellen, die explizit ein Protokoll definieren.

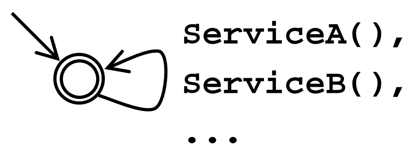


Abbildung 2.12.: Trivial-Protokoll als FSM dargestellt: Dienstaufrufe sind in beliebiger Reihenfolge möglich

Der *Provided Type* definiert keine benötigten Schnittstellen verbindlich. Soll ein solcher Komponenten-Typ substituiert werden oder für eine Substitution verwendet werden, nimmt man auf der *Requires*-Seite implizit genau eine Schnittstelle an. Diese Schnittstelle ist leer, definiert also keine Dienste und ein Protokoll, das nur einen Endzustand besitzt, also die leere Sprache ($L = \emptyset$) akzeptiert. Auf diese Weise kann

anhand der allgemein definierten Regeln überprüft werden, ob eine Substitution gültig ist.

Werden ein *Provided Type* oder ein *Complete Type* durch einen *Implementation Type* substituiert, sollte bedacht werden, dass mit dem *Implementation Type* und dem damit verbundenen SEFF das benötigte Protokoll berechenbar wird. Sollte bereits vor der Substitution manuell ein Protokoll auf der benötigten Schnittstelle definiert worden sein, darf dieses nicht im Widerspruch zum berechneten Protokoll stehen.

2.11.3. Delegations-Konnektoren

Delegations-Konnektoren lassen sich nicht ohne Weiteres mit den in Kapitel 2.11.2 angeführten Regeln auf Validität prüfen, da sie zunächst gleichartige Schnittstellentypen miteinander verbinden. Betrachtet man bei *Provides* Delegations-Konnektoren die äußere Schnittstelle als *Requires*-Schnittstelle und bei *Requires* Delegations-Konnektoren die äußere Schnittstelle als *Provides*-Schnittstelle, können die bereits bekannten Regeln aus Kapitel 2.11.2 angewendet werden.

Auf der *Provides*-Seite darf die äußere Schnittstelle also nur die gleiche Schnittstelle wie die innere anbieten oder diese eingeschränkt anbieten. Dies führt dazu, dass alle Dienstaufrufe auf definierte innere Schnittstellen geleitet werden. Auf der *Requires*-Seite darf die äußere Schnittstelle gleich viele oder mehr Dienste wie die innere Schnittstelle verlangen. Auf diese Weise können von der äußeren Schnittstelle zwar nicht benötigte Anforderungen definiert werden, anders herum laufen jedoch keine Dienstanfragen ins Leere.

2.11.4. Einschränkungen

Ob die in diesem Kapitel genannten Regeln für die Interoperabilität von Komponenten-Typen tatsächlich eingehalten werden, hängt von der konkreten Implementierung des Komponenten-Meta-Modells ab. Zunächst einmal gibt das Komponenten-Meta-Modell, das beschrieben wurde, lediglich einen Satz von allgemeinen Regeln vor, kann die Einhaltung jedoch nicht aktiv überwachen. Implementierungen des Komponenten-Meta-Modells sind jedoch daran gehalten, Interoperabilitätsprüfungen, die nicht zuletzt auch konsistenz-erhaltend sind, durchzuführen.

Es sollte bedacht werden, dass weder für Protokolle noch SEFFs vom Komponenten-Meta-Modell konkrete Umsetzungen (etwa FSMs oder Petri-Netze) vorgegeben werden. Entsprechend müssen die Algorithmen zur Prüfung von Interoperabilität durch eine Implementierung des Komponenten-Meta-Modells austauschbar sein. Zur Umsetzung der Erweiterbarkeit um „Prüfmodule“ bieten sich *Plugins* an, die dann jeweils die Interoperabilitätsprüfung für eine bestimmte Protokoll- oder SEFF-Umsetzung übernehmen.

Allgemeine Bemerkungen zur Validierung von Instanzen des Komponentenmodells finden sich in Kapitel 2.20.

2.12. First Class Entities

First Class Entities (Entitäten erster Klasse) bezeichnen im Komponentenmodell all jene Entitäten, die direkt in einer Modellinstanz existieren können, da sie von keiner

anderen Entität abhängen. Die Modellierung einer Modellinstanz muss mit einer *First Class Entity* beginnen. Zu den *First Class Entities* zählen:

- Komponenten (gleich welcher Ebene)
- Schnittstellen
- Ressourcen

Alle anderen Entitäten hängen von *First Class Enitities* ab und können nicht ohne eine *First Class Entity* existieren. Deshalb werden sie *Second Class Entities* genannt. Beispielsweise kann es keine Signatur ohne eine Schnittstelle geben.

2.13. Benutzer-Rollen

Benutzer- oder Anwender-Rollen sind kein expliziter Bestandteil des Komponentenmodells, gleichwohl wird über Benutzer-Rollen unterschieden, welche Sichten auf das Komponentenmodell möglich sind. In Kapitel 2.14.1 wird auf Benutzer-Rollen Bezug genommen, die wie folgt definiert sind:

- **Entwickler** entwerfen und definieren Komponenten und Schnittstellen sowie Konnektoren zwischen diesen Entitäten. In [43], Seite 2, wird diese Rolle des *Component Developers* näher beschrieben.
- ***System-Deployer*** definieren Beziehungen zwischen Komponenten, Konnektoren und Ressourcen (siehe Kapitel 2.14 und [43], S. 3).

Benutzer-Rollen sind strikt von Rollen (von Komponenten), wie sie in Kapitel 2.4 definiert werden, zu unterscheiden.

2.14. Allokation und Ressourcen

Definition In der Palladio-Gruppe wird der Begriff der *Allokation* verwendet, um die Zuordnung von Entitäten einer Assembly zu Ressourcen zu beschreiben. Die Semantik des Begriffs ist ähnlich dem *Deployment*: In der Allokation wird festgelegt, welche Kontext-Komponente welche Ressource nutzt. Zudem erfolgt eine Abbildung von Assembly Konnektoren auf „Verbindungsressourcen“, also Netzwerk-Verbindungen. Als Stereotyp der Relation zwischen Komponenten, Konnektoren und Ressourcen wird ≪`deployed-on`≫ verwendet.

Allokationen findet immer für Komponenten und Assembly Konnektoren im Kontext statt (in den Abbildungen 2.13 und 2.14 über den Stereotyp ≪`context-comp`≫ gekennzeichnet). Assembly Konnektoren liegen immer im Kontext vor, da sie nur zwischen Kontext-Rollen definiert sind. Dass die *Allokation* nur für Kontext-Komponenten sinnvoll ist, wird schnell ersichtlich, wenn man bedenkt, dass der gleiche Komponenten-Typ sehr oft verwendet werden kann und dabei mit unterschiedlichen anderen Kontext-Komponenten interagiert, die nicht auf den gleichen Ressourcen liegen.

Als Ressourcen-Typen kennt das Komponentenmodell:

- **Rechnende Ressourcen** (*calculating resources*); etwa Server-Hardware. In Abbildung 2.14 werden diese Ressourcen UML2-konform als „Boxen" dargestellt.

- **Nicht-Rechnende Ressourcen** (*non-calculating resources*) / Kommunikations-Ressourcen; hierzu zählen Netzwerkverbindungen, also jene Ressourcen, die zur Datenübertragung zwischen rechnenden Ressourcen vermitteln. In Abbildung 2.14 wird diese Form der Ressourcen UML2-konform als Linie zwischen den berechnenden Ressourcen dargestellt. nicht-rechnende Ressourcen verbinden genau zwei nicht identische rechnende Ressource bidirektional (Daten können in beide Richtungen fließen) miteinander.

Kontext-Komponenten können nur auf rechnenden Ressourcen *deployed* werden. Assembly Konnektoren können nur auf nicht-rechnenden Ressourcen *deployed* werden. Da für den *System Deployer* das Innere von *Composite Components* nicht sichtbar ist, erfolgt das *Deployment* aller Sub-Komponenten einer *Composite Component* stets auf die gleiche rechnende Ressource (siehe auch Kapitel 2.14.3). Umgekehrt können beide Arten von Ressourcen eine beliebige Menge (`0..*`) Komponenten, respektive Assembly Konnektoren, aufnehmen.

Ein *vollständiges* Deployment liegt vor, wenn jede Komponente auf genau einer rechnenden Ressource *deployed* wurde. Zusätzlich muss jeder Assembly Konnektor, der Rollen von Komponenten verbindet, die nicht auf der gleichen rechnenden Ressource liegen, auf genau einer nicht-rechnenden Ressource *deployed* werden. Wird ein Assembly Konnektor nicht *deployed*, weil die Kontext-Rollen an beiden Seiten des Assembly Konnektors Kontext-Komponenten auf der gleichen Ressource zugeordnet werden, entspricht dies einem lokalen Dienstaufruf.

Delegations-Konnektoren können nicht gezielt *deployed* werden. Der *System Deployer* hat keinen Einblick in *Composite Components*, weshalb Delegations-Konnektoren nicht identifiziert werden können.

Je nach verwendeter Komponententechnologie können zusätzliche Einschränkungen auftreten. So werden *Composite Components* in realen Komponentensystemen häufig mit dem *Deployment* faktisch aufgelöst. Lediglich *Basic Components* werden in die Ausführungsumgebung eingebracht. Damit liegen keine Delegations-Konnektoren mehr vor, die deployed werden könnten.

Gültige und ungültige Allokationsformen In Abbildung 2.13 wird gegenüber gestellt, welche Allokationsformen legal sind. Auf der linken Seite wird eine ungültige Variante dargestellt, bei der eine Kontext-Komponente („Component A") auf verschiedene Ressourcen („Server 1" und „Server 2") verteilt werden soll. Auf der rechten Seite erfolgt eine gültige Allokation. Die Kontext-Komponenten „Component B" und „Component C" werden jeweils auf eigene Ressourcen verteilt. Eine ebenfalls gültige Variante wäre die Verteilung der Komponenten B und C auf die gleiche Ressource.

Gleichwohl ist eine Verteilung von „Component A" in veränderter Form möglich. Dazu müssen *zwei* Kontext-Komponenten zum Komponenten-*Typ* von „Component A" definiert werden. Diese Kontext-Komponenten des gleichen Komponenten-Typs können dann auf unterschiedliche Ressourcen verteilt werden.

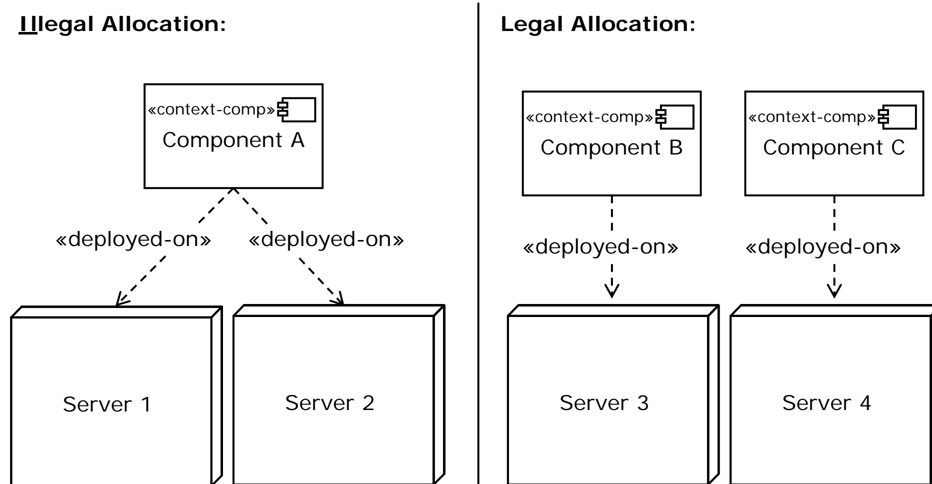


Abbildung 2.13.: Illegale und legale Allokation von Komponenten

2.14.1. Assembly

Eine Menge miteinander verbundener Komponenten, die innerhalb keiner *Composite Component* liegen, stellt im Verständnis des Komponentenmodells eine Assembly dar. Gelegentlich wird die Assembly auch als *Assembly-Composition* bezeichnet. Eine Assembly ist *keine* Entität des Komponentenmodells, sondern stellt lediglich ein *Konzept* zum Erfassen von Komponentenarchitekturen dar. Daher existiert eine Assembly implizit mit der Erstellung einer Komponentenarchitektur.

Im Gegensatz zu .NET-Assemblies oder JAR-Dateien, die möglicherweise mit Assemblies im Verständnis des Komponentenmodells assoziiert werden könnten, verbietet das Komponentenmodell eine erneute Komposition von Komponenten oder anderer Assemblies aus bestehenden Assemblies. Assemblies im Sinne des Komponentenmodells stellen also eine abgeschlossene Komponentenarchitektur dar.

An dieser Stelle sollen die Eigenschaften zusammengefasst werden.

- Eine Assembly fasst eine Menge von Kontext-Komponenten (`1..*`) und die dazugehörigen „äußeren" Assembly Konnektoren zusammen. Als „äußere" Assembly Konnektoren gelten dabei jene, die innerhalb keiner Composite Component liegen.

- Die Kontext-Komponenten im Inneren einer Assembly sind nicht auf Kontext-Instanzen einer bestimmten Komponenten-Typ-Ebene beschränkt. Zur gleichen Zeit dürfen auch Kontext-Komponenten unterschiedlicher Typ-Ebenen auftreten.

- Erst eine Assembly ermöglicht es, dass es Assembly Konnektoren zwischen Komponenten gibt, die innerhalb keiner *Composite Component* liegen.

- Assemblies werden von *System-Deployern* verwendet. Die für diese Zielgruppe definierte Sicht blendet dabei das Innere von *Composite Components* aus (*Black-Box*). In der Folge kann ein *System-Deployer* bei der Allokation *Composite Components* nicht auf verschiedene Ressourcen verteilen, da diese Komponente keine für ihn identifizierbaren Sub-Komponenten besitzt, die sich einzeln ansprechen ließen.

- Für Komponenten-Entwickler (*System Architects*) ist das Innere von *Composite Components* sichtbar. Komponenten-Entwicklern steht mit Assemblies ein Mittel zur Verfügung, mit dem sie festlegen können, wie eine Komponentenarchitektur realisiert werden soll. Eine Assembly stellt damit eine Gesamtarchitektur und ein Softwaresystem dar, die durch Entwickler erzeugt werden.

Zu einem Zeitpunkt kann nur genau eine Assembly im Modell allokiert werden. Pro Ausführungsumgebung kann genau eine Assembly oder ein Teil einer Assembly (bei Verteilung über verschiedene Ressourcen) allokiert werden.

2.14.2. Vergleich von Allokation und Assembly

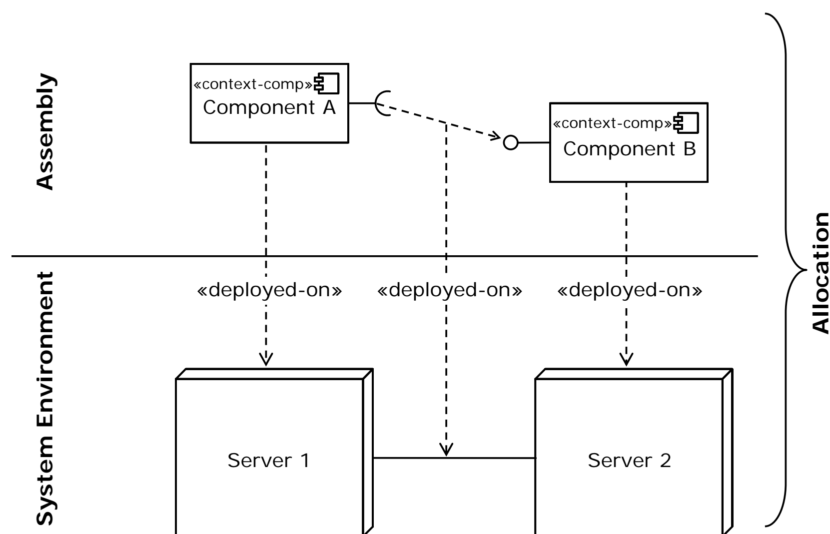


Abbildung 2.14.: Allokation einer Assembly in einer Systemumgebung (nach [10], S. 26)

Abbildung 2.14 zeigt beispielhaft die Allokation einer Assembly, bestehend aus zwei Kontext-Komponenten und einem Assembly Konnektor, auf zwei Server sowie eine Verbindung zwischen den Servern. Dabei wird zwischen der Begrifflichkeit der Assembly und der Allokation (*Allocation*) unterschieden, so wie sie im Kontext des Palladio Komponentenmodells verwendet werden. Die Allokation beschreibt die Verteilung auf Ressourcen, wohingegen eine Assembly eine Menge von Komponenten und die damit assoziierten Konnektoren zusammenfasst. Komponenten und Konnektoren einer Assembly werden über die ≪`deployed-on`≫-Relation allokiert, sind also Teil der Allokation.

Zur Unterscheidung von Allokation (*Deployment*) und Assembly ist der Zeitpunkt wichtig, zu dem die Aktion vorgenommen wird:

- Die **Assembly** wird zum Zeitpunkt der Konstruktion über ein System bzw. eine Komponentenarchitektur und durch den *System Architect* definiert.

- Die **Allokation** wird zum Zeitpunkt der Installation durch den *System Deployer* vorgenommen.

Die Assembly-Definition kann unabhängig von der Allokation erfolgen.

Nach der in einer Assembly definierten Komposition richten sich die tatsächlich aufgerufenen und aufrufenden Dienste einer Komponente. In einer Assembly wird der *Kontext* (vgl. Kapitel 2.16) festgelegt. Damit können z. B. die Aufrufparameter von Diensten einer Komponente auf Grund der konkreten Verwendung variieren, weil die aufrufende Komponente lediglich ein begrenztes Spektrum der möglichen Aufrufparameter nutzt. Mit der Verwendung (und „Verdrahtung") einer Komponente in einer Assembly lassen sich *funktionale Eigenschaften* einer Komponente bestimmen. Zugleich hat die „Verdrahtung" von Komponenten einen starken Einfluss auf *nicht-funktionale Eigenschaften*. So hängen viele nicht-funktionale Eigenschaften einer Komponente maßgeblich von den Komponenten ab, die die benötigten Schnittstellen erfüllen. Sind letztere Komponenten beispielsweise sehr wenig zuverlässig, hat dies Auswirkungen auf die betrachtete Komponente.

Mit der Allokation variiert unter anderem die Ausführungsumgebung einer Komponente. Dies hat z. B. Auswirkungen auf die Ausführungsgeschwindigkeit, Sicherheit oder die Konfiguration zur Ausführung einer Komponente. Vornehmlich sind hiervon also *nicht-funktionale Eigenschaften* betroffen.

2.14.3. Assembly und Composite Component

In der Sprachwelt des Komponentenmodells sind *Assemblies* und *Composite Components* strikt zu unterscheiden (nach [10], S. 27f).

Composite Component Wie bereits beschrieben, setzt sich eine *Composite Component* aus anderen Komponenten zusammen. Zum Zeitpunkt des *Deployments* ist das Innere, die Realisierung einer *Composite Component* eine *Black-Box*. Daher ist es einem *System Deployer* nicht möglich, eine *Composite Component* über verschiedene Ausführungsumgebungen (*Execution Environment*) zu verteilen.

Assembly Auch *Assemblies* fassen eine Menge von Komponenten zusammen. Im Gegensatz zur *Composite Component* ist das Innere einer Assembly jedoch zum Zeitpunkt des *Deployments* für den *System Deployer* sichtbar (*White-Box*). Dies ermöglicht, dass innere Komponenten einer Assembly auf unterschiedlichen Ausführungsumgebungen allokiert werden können.

Assembly vs. Komponente Eine Assembly ist keine Komponente. Daher kann sie nach außen keine Dienste über die bislang dargestellten Schnittstellen erbringen. Sie fasst lediglich eine Menge von Komponenten und Konnektoren zusammen. Dennoch lassen sich Assemblies um Komponenten oder mit weiteren Assemblies ergänzen, indem die zusätzlichen Komponenten und Konnektoren mit in die Assembly aufgenommen werden.

Assemblies können hingegen Systemschnittstellen haben, auf denen sie System-Dienste erbringen. Diese Form der Schnittstellen können im Komponentenmodell derzeit nicht modelliert werden. Daher wird auf Systemschnittstellen im Weiteren nicht näher eingegangen.

Abbildung 2.15 beschreibt die Umwandlung einer Assembly in eine *Composite Component*, so wie der Vorgang von einem *System Deployer* wahrgenommen würde. Da das

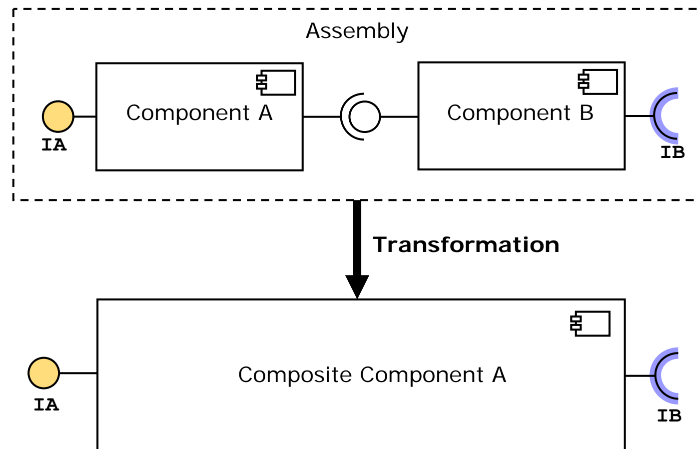


Abbildung 2.15.: Transformation einer Assembly in eine *Composite Component* (aus Sicht eines *System Deployers*)

Innere einer *Composite Component* nicht mehr sichtbar ist, würden innere Komponenten und Konnektoren bei einer Transformation unsichtbar.

Für eine solche Transformation muss angegeben werden, welche Schnittstellen von der neu definierten *Composite Component* angeboten und benötigt werden, damit die neue *Composite Component* vollständig spezifiziert ist. Ebenso müssen Delegations-Konnektoren definiert werden, auch wenn sie für den *System Deployer* nicht sichtbar sind, damit alle Aufrufe definierten Aufrufpfaden folgen können.

2.15. Komponenten-Typ vs. Komponenten-Verwendung

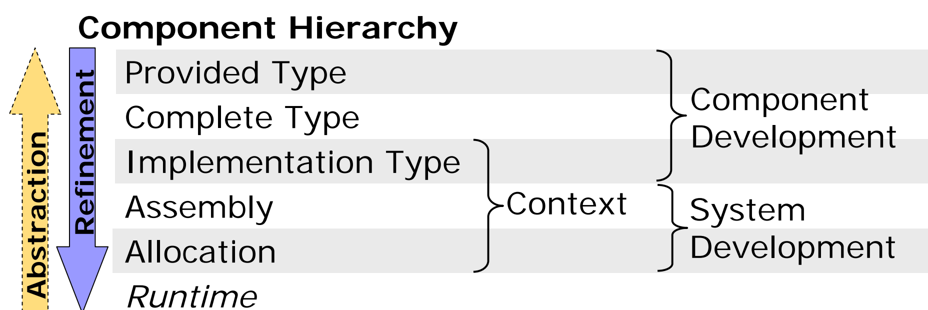


Abbildung 2.16.: Komponentenhierarchie mit Einordnung des Kontexts (nach [10], S. 33)

In Abbildung 2.16 werden alle Ebenen des Komponentenmodells aufgezeigt. Wichtig ist die Unterscheidung der obersten drei Ebenen von der vierten (Assembly) und fünften (Allokation) Ebene.

- *Provided Type*, *Complete Type* und *Implementation Type* definieren jeweils Komponenten-Typen auf verschieden Detail-Niveaus („Component Development“).

Kompositions-Strukturen sind nur innerhalb von Komponenten möglich. In Analogie zur objektorientierten Programmierung, werden Typen über Klassen festgelegt, die vorgegebenen Regeln folgen. Betrachtet man die Erstellung und Verfeinerung von Komponenten-Typen als Prozess, so liegt am Ende des Prozesses eine ungeordnete Menge von Komponenten vor, die einer Verwendung harren. Die Menge definierter Komponenten-Typen kann als Komponenten-*Repository* interpretiert werden.

- Assembly und Allokation sowie das Innere von *Implementation Types* (als *Composite Component* realisiert) erlauben es hingegen, Kompositions-Strukturen zwischen Komponenten zu definieren („System Development“). Komponenten-Typen können über die Assembly und Allokation „verwendet“ werden. In Analogie zum Klassen-Beispiel werden die Komponenten-Typen zu Objekt-Instanzen. Die zuvor definierten Komponenten-Typen werden verwendet („Kontext-Komponente“) und über Assembly Konnektoren untereinander verbunden. Damit werden auf diesen Ebenen Komponentenarchitekturen definiert.

2.16. Kontext

Im vorigen Kapitel wurde die *Verwendung* von Komponenten-Typen gegenüber der *Definition* von Komponenten-Typen abgegrenzt. Als Kurzdefinition wurde der Begriff des *Kontexts*, der in diesem Zusammenhang wichtig ist, bereits eingeführt. An dieser Stelle erfolgt eine vollständige Darstellung des Kontexts unter Rückgriff auf die bis jetzt eingeführten Konzepte des Komponentenmodells.

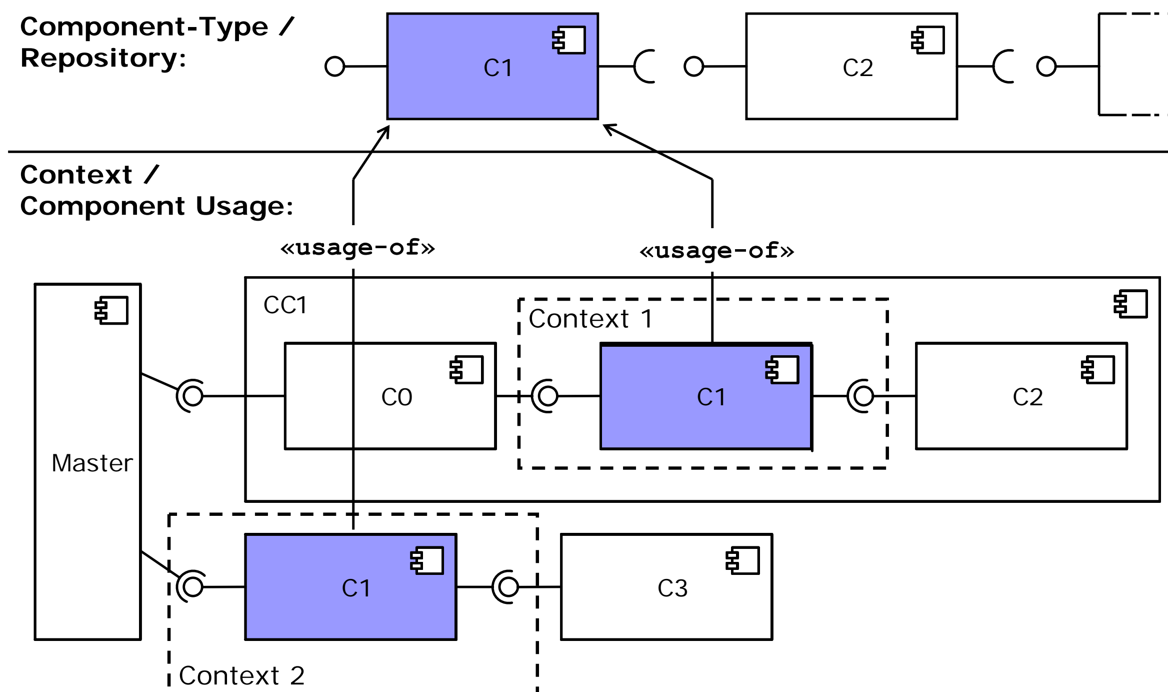


Abbildung 2.17.: Unterscheidung von Komponenten-Typ und Komponenten-Kontext

Idee Verglichen mit Vorversionen des Palladio Komponentenmodells, stellt der Kontext (*Context*) das neueste Konzept dar. Abbildung 2.17 zeigt, wie sich ein Kontext definiert. Im Beispiel kommt der Komponenten-Typ „C1“ in zwei unterschiedlichen Kontexten vor. Zum einen (in „Context 1“) ist sie mit „C0“ auf der *Provides*-Seite und „C2“ auf der *Requires*-Seite verbunden, zum anderen (in „Context 2“) ist sie auf der *Provides*-Seite mit „Master“ und auf der *Requires*-Seite mit „C3“ verbunden. Um exakt definieren zu können, welche Verwendung von „C1“ bzw. eines Komponenten-Typs im Allgemeinen gemeint ist, wird der Kontext eingeführt. Dieser beschreibt, wie ein Komponenten-Typ verwendet wird, also mit welchen anderen Komponenten eine Komponente verbunden ist. Der Kontext identifiziert eindeutig den Ort der Verwendung eines Komponenten-Typs. Komponenten-Typen, die in Kontexten verwendet werden, heißen *Kontext-Komponenten*. Zu einem Komponenten-Typ kann es `0..*` Kontext-Komponenten geben.

Der gerade skizzierte Begriff des Kontexts bezieht sich auf das Vorkommen von Kontext-Komponenten in Assemblies und Kompositionsstrukturen. Neben dieser Bedeutung definiert das Komponentenmodell den Kontext ebenfalls für Allokationen. Wird eine Kontext-Komponente aus der Allokationssicht betrachtet, liegt die Kontext-Komponente in einem Kontext, der auch die verwendeten Ressourcen erfasst.

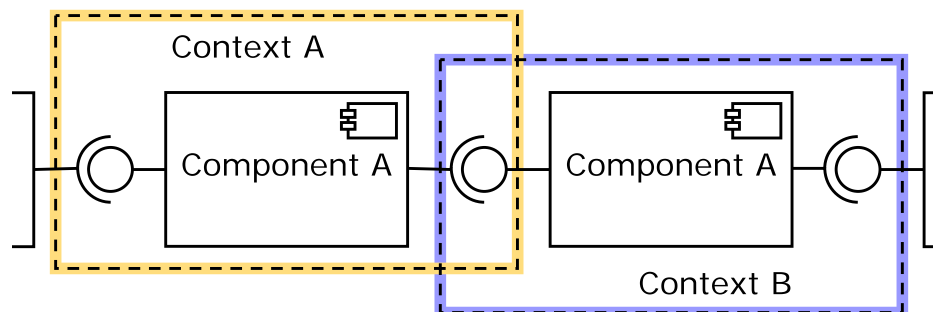


Abbildung 2.18.: Komponente-Typ „Component A“ in zwei unterschiedlichen Kontexten

In Abbildung 2.18 wird „Component A“ zweifach verwendet. Da es sich um den gleiche Komponenten-Typ handelt, ließen sich die Verwendungen nicht unterscheiden. Über den Kontext gelingt jedoch die Unterscheidung. Die linke Kontext-Komponente liegt in „Context A“, die rechte Kontext-Komponente in „Context B“. Da der Kontext implizit vorhanden ist, bleibt die mehrfache Verwendung des gleichen Komponenten-Typs in jedem Fall eindeutig.

Impliziter Kontext Komponenten-Typen können nie ohne Kontext verwendet werden. Wichtig ist zu beachten, dass Komponenten-Typen *in* einem Kontext liegen. Zu jedem Komponenten-Typen, der außerhalb des *Repositories* verwendet wird, gibt es *implizit* ∃! Kontext. Da jeder Kontext eindeutig ist (siehe Kapitel 2.18), kann ein Komponenten-Typ – beispielsweise auf der Assembly-Ebene – auf der gleichen Hierarchie-Stufe mehrfach verwendet werden, die Verwendung bleibt automatisch eindeutig über den Kontext.

Kontext-Rollen Im Kapitel über Konnektoren wurde definiert, dass Konnektoren stets *Kontext*-Rollen verbinden. Bisher wurden Kontext-Rollen noch nicht näher be-

trachtet. Kontext-Rollen unterscheiden sich von „normalen“ Rollen lediglich durch den Ort ihrer Verwendung. Komponenten-Typen sind mit „normalen“ Rollen assoziiert. Kontext-Komponenten sind mit Kontext-Rollen assoziiert. Der Kontext, in dem eine Kontext-Komponente und ihre Kontext-Rollen liegen, muss der gleiche sein. Der Kontext wird durch die Kontext-Komponente bestimmt. Alle assoziierten Kontext-Rollen einer Kontext-Komponente fallen automatisch in den gleichen Kontext wie die Kontext-Komponente.

Eine Kontext-Komponente muss die gleichen Kontext-Rollen anbieten und benötigen, wie der Komponenten-Typ „normale“ Rollen anbietet und benötigt. Zwischen „normalen“ Rollen und Kontext-Rollen gibt es ebenso eine „usage-of“ Beziehung wie zwischen Komponenten-Typen und Kontext-Komponenten. Kontext-Rollen sind Instanzen „normaler“ Rollen in einem Kontext.

Um die Unterscheidung zwischen *Provided Roles* und *Required Roles* zu erhalten, gibt es *Provided* Kontext-Rollen und *Required* Kontext-Rollen.

Identifikation Wie in Abbildung 2.16 zu sehen ist, kann ein Kontext unterschiedliche Detail-Niveaus erfassen. Er ist definiert für:

- **Implementation**. Es werden die Kompositionsstrukturen innerhalb von *Composite Components* eindeutig. Da die inneren Assembly Konnektoren von Composite Components zwischen Kontext-Rollen verbinden, erfolgt eine Verbindung über zwei eindeutige Kontexte. Eindeutig im Kontext sind somit: Kontext-Komponenten, Kontext-Rollen, Delegations-Konnektoren und Assembly Konnektoren.

- **Assembly**. Auf diesem Detail-Niveau werden die Kompositionsstrukturen der Komponentenarchitektur eindeutig, da auch außerhalb von *Composite Components* liegende Assembly Konnektoren erfasst werden.

- **Allokation**. Hier werden zusätzlich die Ressourcen (*computational*, *non-computational*) über die ≪`deployed-on`≫-Relation und die Ausführungsumgebung eindeutig erfasst, auf denen die Kontext-Komponenten und Konnektoren allokiert werden.

  Über Kontexte auf der Allokationsebene lassen sich zusätzliche Eigenschaften wie *Concurrency*, *Security*, Container-Eigenschaften und Komponentenkonfigurationen erfassen.

Das Ziel des Konzepts des Kontexts ist die Schaffung einer Möglichkeit zur Unterscheidung von Komponenteneigenschaften je nach ihrem konkreten Verwendungskontext. So kann der *gleiche* Komponenten-Typ ein unterschiedliches Verhalten je nach Kontext aufweisen, je nachdem, von welcher Komponente die benötigten Dienste einer Komponente erfüllt werden. Wird eine wenig performante, dafür aber hochgradig zuverlässige Komponente verwendet, oder eine hoch performante und wenig zuverlässige Komponente verwendet, variieren die QoS-Werte einer nutzenden Komponente stark. Vorhersagen oder Berechnungen von Komponenteneigenschaften können also erst genauer durchgeführt werden, wenn der Komponenten-Kontext in die Berechnungen einfließt.

Graphische Darstellung In den graphischen Darstellungen dieses Kapitels wurden die Kontexte hervorgehoben eingezeichnet. Der Kontext sollte jedoch als immanente Eigenschaft von Kontext-Komponenten und auch Kontext-Rollen verstanden werden, der nur zur einfacheren Erkennbarkeit des Kontexts gezeichnet wird.

2.17. Hierarchie-Ebenen

Abbildung 2.16 (Seite 48) erfasst die bereits in vorangehenden Kapiteln definierten Komponenten-Typ-Ebenen, die Assembly, die Allokation (*Allocation*) und die Laufzeit (*Runtime*) in einer Graphik. Letztere Ebene ist im Komponentenmodell bisher lediglich angedacht, jedoch noch nicht umgesetzt. Weitere Informationen hierzu finden sich in Kapitel 2.22, Seite 64.

Je tiefer eine Ebene in der Graphik dargestellt ist, desto detaillierter können Komponenten-Typen definiert werden, entsprechend sind mehr Attribute über die Komponenten auf den niederen Ebenen definiert. In umgekehrter Richtung nach oben werden die Komponenteninformationen abstrakter und kommen damit einer Grob-Modellierung von Komponentenarchitekturen entgegen.

2.18. Identität

Alle Entitäten des Komponentenmodells tragen einen eindeutigen Bezeichner (ID). Werden in Relationen andere Entitäten referenziert, erfolgt diese Referenzierung stets über die IDs der Entitäten, damit an keiner Stelle die Relationen des Komponentenmodells mehrdeutig sein können.

Bieten beispielsweise zwei Komponenten mit dem gleichen Namen „Component“ die gleiche Schnittstelle „InterfaceP“ an und benötigen gleichermaßen die Schnittstelle „InterfaceR“ sind sie als *Complete Type* über keine weiteren Merkmale zu unterscheiden. Daher werden sie zusätzlich mit einem eindeutigen Bezeichner / ID versehen, um unterscheidbar zu sein.

Eine Abfrage auf Gleichheit (*equals*) zweier Entitäten wird im Komponentenmodell auf IDs zurückgeführt. Sind zwei IDs identisch, handelt es sich um die gleiche Entität. Die Gleichheit muss strikt von der Prüfung auf Konformität, wie sie in Kapitel 2.11 beschrieben wird, unterschieden werden. Gleiche Typen sind zu einander konform, aus der Konformität kann jedoch keine Gleichheit gefolgert werden.

Um grundsätzlich die Gleichheit zwischen zwei Entitäten unterschiedlichen Typs (etwa *Provided Component Type* und *Interface*) auszuschließen, sind IDs im Komponentenmodell Tupel, die aus einem eindeutig identifizierenden Teil („ID-Teil“) sowie einem ID-Typ bestehen. Je Typ von Entität gibt es einen ID-Typ. Das bedeutet, dass es *Provided Component Type*-IDs, *Interface*-IDs usw. gibt. Damit können selbst zwei Komponenten unterschiedlicher Typ-Ebenen mit dem gleichen ID-Teil über den ID-Typ unterschieden werden und vor allem nie identisch sein. Zwei IDs sind nur gleich, wenn sowohl der ID-Teil als auch der ID-Typ gleich sind.

Um den ID-Teil auch bei Zusammenführung unterschiedlicher Komponentenmodell-Instanzen eindeutig zu erhalten, wird empfohlen, ein der *Globally Unique Identifier* (GUID, siehe auch [77]) ähnliches Konzept zu verwenden. Dieses Konzept macht es

auch über Systemgrenzen hinweg und für einander vollkommen unbekannte Systeme wahrscheinlich, dass generierte GUIDs eindeutig sind.

In Implementierungen des Komponenten-Meta-Modells muss sich das ID-Konzept wiederfinden. Entitäten müssen also stets über eine Entsprechung zum eindeutigen Bezeichner verfügen. IDs können intern verwaltet werden und somit der Kontrolle eines Modellierers entzogen werden, damit die Eindeutigkeit durch eine Implementierung sichergestellt werden kann.

Die folgenden Entitäten (respektive Typen von Entitäten) tragen IDs; die ID-Typen unterscheiden sich entsprechend:

- Kontext. Jeder Kontext trägt eine eigene ID. Zu jeder Verwendung einer Komponente entsteht ein Kontext mit neuer ID. Kontext-IDs können unter keinen Umständen mehrfach auftreten.
- Komponenten-Typen (je Typ-Ebene ein eigener ID-Typ). Jeder Komponenten-Typ einer Typ-Ebene lässt sich damit von anderen unterscheiden. Wird ein Komponenten-Typ verfeinert, so tragen auch jene Komponenten, die untereinander in einer *conforms*-Beziehung stehen, unterschiedliche IDs. Jede Kontext-Komponente hat eine eindeutige ID.
- Schnittstellen. Jede Schnittstelle unterscheidet sich von jeder anderen über die ID, auch wenn die Schnittstellen ansonsten vollkommen identisch sind (bspw. exakt die gleichen Signaturen tragen – aber etwa eine abweichende Semantik besitzen).
- Konnektoren. Die IDs von Konnektoren müssen eindeutig sein.
- Rollen. Jede Rolle hat eine eigene ID. Jede Kontext-Rolle hat eine von der „normalen“ Rolle und untereinander verschiedene ID.
- Ressourcen. IDs von Ressourcen müssen sich unterscheiden, wenn die dahinter stehende Hardware oder Ausführungsumgebung eine andere ist.
- Annotationen. Annotationen selbst können über IDs referenziert werden. Zudem lassen sich Datentypen über ID identifizieren. Siehe hierzu auch Kapitel 2.19.

2.19. Annotationen

Das Komponentenmodell bietet die Möglichkeit alle Entitäten des Modells mit beliebigen Zusatzinformationen „Annotationen“ (Attribute) zu versehen. Da die genauen Anwendungsszenarien des Komponentenmodells nicht exakt absehbar sind und Vorgaben das Modell nicht unnötig limitieren sollen, bleibt das Komponentenmodell, ebenso wie bei Protokollen und SEFFs, offen und gibt keine feste Menge von Annotationen vor. Damit lassen sich beliebige Annotationen ergänzen, sofern sie benötigt werden. Dies können ebenso Messwerte wie *Quality of Service* Merkmale (siehe Kapitel 2.9.2) von Entitäten, komplexe Datentypen wie Verteilungsfunktionen oder beliebige andere Annotationen sein. Zusätzlich sind ebenfalls *Constraints* denkbar, die über Validierungsalgorithmen (Kapitel 2.20) erfasst werden. Das heißt also, dass weder die möglichen Semantiken, die Datentypen noch Datenstrukturen einer Annotation durch das Komponentenmodell abschließend festgelegt werden.

Gleichwohl definiert das Komponentenmodell einen Standardsatz von Annotationen, die eine schnelle Arbeit mit dem Komponentenmodell vereinfachen sollen. Hierzu zählen Annotationen die die Datentypen Integer (`int`, `long`), Gleitkommazahl (`float`, `double`) und Zeichenkette (`string`) analog ihrer Definition in .NET C# 1.1 (siehe [51, 52, 53]) besitzen.

Der gleiche Datentyp einer Annotation kann damit unterschiedliche Semantiken aufweisen. Um eine eindeutige Zuordnung einer Semantik zu einer Annotation zu ermöglichen, muss jede Annotation ihren Annotations-Typ definieren. Dazu gibt es eine Liste eindeutiger Annotations-Typen, die eine feste Verknüpfung zwischen Annotations-Typ und Semantik erlaubt. Damit besitzen Annotationen ein Identifizierungsmerkmal, das auch über mehrere Instanzen von Annotationen hinweg benutzt werden kann, um beispielsweise eine speziellen Laufzeitmesswert bei allen Komponenten einer Modell-Instanz zu kennzeichnen.

Annotation können überdies aus anderen Annotationen berechnet werden. Um eine Unterscheidung zu erlauben, welche Werte berechnet wurden, gibt es je Annotation ein *Calculated-Flag*, das diese Werte kennzeichnet. Im Allgemeinen sind berechnete Annotationen *read-only*. Gleichwohl können Berechnungen, die auf eine Komponentenmodellinstanz durchgeführt wurden, berechnete Werte setzen.

Es werden immer Instanzen von Annotationen Instanzen von Entitäten des Komponentenmodells zugeordnet. Annotationen können zu den folgenden Entitäten des Komponentenmodells vergeben werden:

- Komponenten, wobei die Annotation eindeutig einer Komponente zugeordnet werden kann.
- Schnittstellen
- Kontexte. Hier tragen die Kontexte Zusatzinformationen (beispielsweise QoS-Informationen), die sich nicht auf einen Komponenten-Typ beziehen, sondern auf Kontext-Komponenten in einer „Umgebung“.
- Assembly Konnektoren. Hier sind im Zusammenhang mit der Abbildung auf Ressourcen beispielsweise Durchsatzwerte als Annotation denkbar.
- Delegations-Konnektoren. Auf der Assembly- und Allokations-Ebene können bspw. Informationen zur Realisierung von Factories (vgl. [34], S. 171ff) einer *Composite Component* hinterlegt werden.
- Ressourcen (berechnend und nicht-berechnend)

Zu einer Entität sind `0..*` Annotationen möglich. Damit kann die gleiche Annotation jedoch nicht für mehrere Entitäten zugleich verwendet werden. Die Annotationen zu einer Entität sind ungeordnet. Annotationen können für bestimmte Entitäten nicht vorgeschrieben werden. Es ist also nicht vorhersehbar, ob eine Entität eine bestimmte Annotation trägt.

Zusätzlich können auch SEFFs und Protokolle Annotationen verwenden. Da die Typen für SEFFs und Protokolle nicht festgelegt sind, kann an dieser Stelle jedoch nicht beschrieben werden, zu welchen dann erweiterten Modellkonstrukten Annotationen vergeben werden können. Annotationen folgen dennoch dem gleichen, oben beschriebenen, Aufbau.

Alternativen Annotation können derzeit nur genau einer Entität zugeordnet werden. Für das Komponentenmodell sind für die Zukunft zwei Alternativen nicht ausgeschlossen:

1. Annotationen sind weiterhin nur genau einer Komponente zugeordnet, können sich jedoch rekursiv auf untergeordnete Entitäten vererben. Mit der Erweiterung würden sich Annotationen einer *Composite Component* auf alle inneren Komponenten vererben.

2. Annotationen könnten alternativ zu `n` Entitäten zugelassen werden. Damit könnten sich unterschiedliche Entitäten die gleichen Annotationen teilen. Zudem ermöglicht diese Variante die Definition globaler Werte, die für eine gesamte Komponentenmodell-Instanz gelten. Dies kann von konkreten Ausfallwahrscheinlichkeiten bis hin zu globalen Zeit-Multiplikatoren reichen.

Beiden Ansätzen ist gemein, dass mehrfach benötigte Werte nicht redundant vorgehalten werden müssen. Damit werden zusätzlich Inkonsistenzen zwischen Annotationen verhindert.

2.20. Validierung von Modellinstanzen

Zunächst einmal muss im Komponentenmodell zwischen *Validierung* und *Konsistenz* unterschieden werden. Das Komponentenmodell muss immer konsistent gehalten werden. Mit Konsistenz ist dabei die Einhaltung gültiger Modellkonstrukte und Modellstruktur gemeint. Würde eine Provided Role zwei Schnittstellen mit einander verknüpfen, wäre dieses ein ungültiges Modellkonstrukt und würde die Konsistenz des Modells verletzten (siehe Abbildung 2.19). Unter der Validierung wird in diesem Zusammenhang eine über die Konsistenzwahrung hinaus gehende Prüfung des Modells verstanden. Validierungen umfassen minimal die Überprüfung auf Gültigkeit der Bedingungen aus den Kapiteln 2.10 und 2.11.

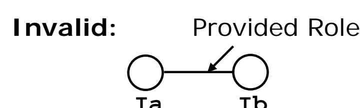


Abbildung 2.19.: Invalides Modellkonstrukt: Konsistenzverletzung

Validierungsalgorithmen sind kein zwingender Bestandteil einer Implementierung des Komponentenmodells, sondern optionale Elemente. Vielmehr stellen verschiedene Formen von Validierungen Strategien dar. Das Komponentemodell lässt gemäß des *Strategy-Patterns* (vgl. [34], S. 373ff) beliebige Definitionen von Validität zu. Über das Komponentenmodell werden lediglich die zu validierenden Strukturen bereitgestellt, womit das Komponentenmodell von den Validierungen gänzlich unabhängig ist.

Zwei grundsätzliche Strategien zur Validierung sind für das Komponentenmodell vorgesehen sind:

- **Permanente Validierung**. Bei dieser Form der Validierung wird eine valider Modellzustand permanent durch Überprüfung eingehalten. Die Überprüfung erfolgt *on-the-fly* bei jeder Modelländerung. Nur Modellelemente, die als valide betrachtet werden, werden zugelassen.

- **Punktuelle Validierung**. Diese Form der Validierung wird stets manuell initiiert und bezieht sich auf einen festen Zustand einer Modellinstanz. Werden invalide Modellelemente erkannt, so erfolgt eine Rückmeldung über die Art des Fehlers und die betroffenen Modellelemente, so dass beispielsweise eine Korrektur vorgenommen werden kann.

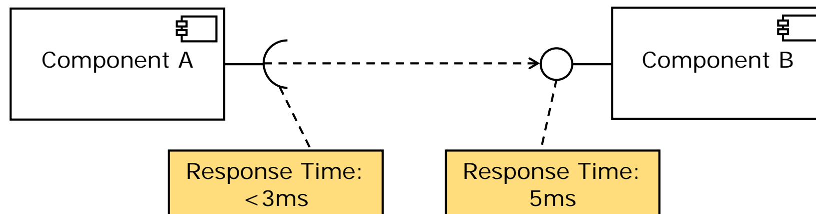


Abbildung 2.20.: Beispiel: Annotation von validitätsrelevanten Informationen

Neben einer Konsistenz-Wahrung, wie sie oben beschrieben wurde, sind beliebige weitere Definitionen von Validität denkbar, die eine strengere Prüfung auf einer Modellinstanz vornehmen. So lassen sich Annotationen in die Validitätsprüfung mit einbeziehen. Dadurch können beispielsweise die annotierte Schnittstellen zu beiden Seiten eines Assembly Konnektors überprüft werden. Liefert die angebotene Schnittstelle nicht mindestens die geforderten Performanz-Werte (über Annotationen definiert), wie in der benötigten Schnittstelle definiert, ist die Modellinstanz invalide. Ein solches Beispiel wird in Abbildung 2.20 dargestellt.

Da über die Validierung auch die Typ-Substitution reglementiert werden kann, können verschiedene „Interoperabilitätsniveaus“ zwischen Komponenten definiert und betrachtet werden. Auch die klassischerweise (vgl. [12], S. 71) betrachteten Niveaus Signatur / Protokoll / QoS werden über Validierungen modellierbar.

Ebenso sind mehrere Definitionen von Validität, die zur gleichen Zeit angewendet werden, denkbar, sofern diese keine widersprüchlichen Anforderungen an eine Modellinstanz stellen. Es gibt nicht *die* valide Instanz des Komponentenmodells, bewusst sind die Validierungen erweiterbar gehalten.

Auch invalide Modellinstanzen des Komponentenmodells müssen serialisiert werden können. Inkonsistente Modelle werden hingegen gar nicht unterstützt und dürfen nicht „erzeugbar“ sein.

2.21. Einschränkungen

Wie jedes Modell stellt das Palladio Komponentenmodell eine Abstraktion realer Komponentenarchitekturen dar. Dies impliziert Einschränkungen der Modellkonstrukte. Das Komponentenmodell ist somit nicht in der Lage beliebige Eigenschaften realer Komponentenarchitekturen abzubilden. Bei diesen Einschränkungen ist zu unterscheiden zwischen jenen Einschränkungen, die im Rahmen der Abstraktion bewusst vorgenommen wurden und Einschränkungen, die in nicht bedachter Weise auftreten und die Nutzbarkeit des Komponentenmodells unerwünscht einschränken.

Generell gilt, dass nur ein gewisses Maß Einschränkungen für Modelle zwingend erforderlich ist, damit Modelle berechenbar bleiben, auf Modellen gerechnet werden kann

und die Zahl der Modellkonstrukte überschaubar und damit verständlich ist. Abstraktion darf nicht zu Gunsten eines allumfassenden und absolut vollständigen Modells aufgegeben werden. Ein solches Modell würde die Vorteile des Modell-Gedankens zu nichte machen.

2.21.1. Zustandsmodellierung

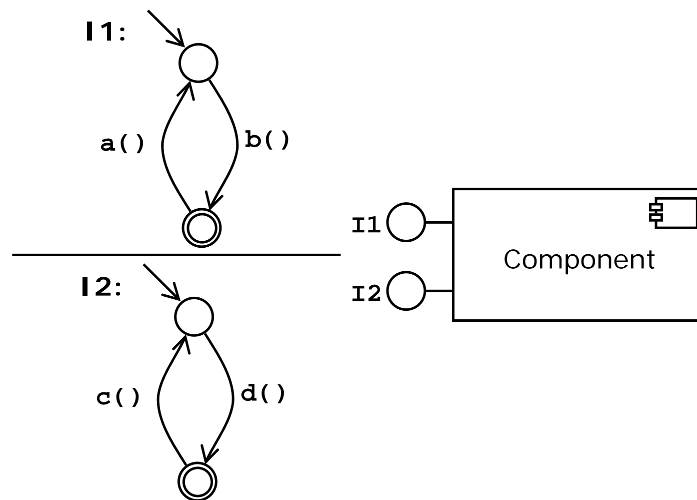


Abbildung 2.21.: Komponente mit zwei angebotenen Schnittstellen und den darauf spezifizierten Protokollen.

Das Schnittstellen-Protokoll wird auf Schnittstellen selbst definiert. Die Zustände eines angebotenen Protokolls entsprechen jedoch internen Zuständen der anbietenden Komponente. Abbildung 2.21 verdeutlicht eine Einschränkung des Komponentenmodells, die die fehlende Synchronisation von Protokollen über angebotene Schnittstellen einer Komponente hinweg betrifft. Je angebotener Schnittstelle hat eine Komponente einen eigenen internen Zustand. Die internen Zustände einer Komponente lassen sich jedoch nicht synchronisieren, die Protokolle werden damit vollkommen unabhängig voneinander durchlaufen.

Im Beispiel können `a()` und `b()` beliebig abwechselnd ausgeführt werden. Auch `c()` und `d()` lassen sich abwechselnd ausführen. Da beide Protokolle jedoch auf unterschiedlichen Schnittstellen (`I1` und `I2`) definiert sind, ließe sich nicht beschreiben, dass nach dem Ausführen von `a()` zunächst `c()` ausgeführt werden müsste.

Eine Komponente hat damit keinen einzelnen eindeutigen Zustand, sondern je nach Anzahl der angebotenen Schnittstellen eine Menge von Zuständen, die zu einem komplexen Komponentenzustand führen.

Funktional ist dies jedoch keine echte Einschränkung. Modelliert man statt einzelner kleiner (gemessen an der Zahl der definierten Dienste) Schnittstellen eine große Schnittstelle, die in der Summe alle Dienste der kleinen Schnittstellen vereint, so resultiert dies in einer Komponente mit nur einem internen Zustand, der über *ein* Protokoll beschrieben werden kann. Damit lassen sich beliebige Abhängigkeiten zwischen Diensten einer Komponente definieren. Bei der Verwendung dieser einen großen Schnittstelle, würde jede benutzende Komponente dann „ihren“ ursprünglichen Teil der Dienste aufrufen und das verbleibende Angebot auf der angebotenen Schnittstelle ignorieren.

Man kann argumentieren, dass Abhängigkeiten von Diensten untereinander darauf hindeuten, dass diese Dienste ohnehin auf *einer* Schnittstelle zu definieren wären, da zwischen unabhängigen Diensten kein Protokoll notwendig wäre. Die Notwendigkeit ein Protokoll definieren zu können, deutete in diesem Fall auf zusammenhängende Funktionalität hin. Eine Aufteilung zusammenhängender Funktionalität auf verschiedene Schnittstellen widerspräche einem guten Schnittstellen-Entwurf.

Umgekehrt betrachtet kann eine große Komponente vielfältige und sehr unterschiedliche Schnittstellen anbieten. Eine Schnittstelle mit „An/Aus“-Funktionalität könnte dennoch ungewollt eine einzelne große Schnittstelle notwendig machen, damit ein gemeinsames Protokoll definiert werden könnte. Aus dieser Sicht würde die einzelne große Schnittstelle das Design der Komponente erheblich einschränken und verschlechtern. Nahezu voneinander unabhängige Funktionalität würden unerwünscht vermischt. Die hinter der Bündelung von Diensten in einer Schnittstelle stehenden Design-Entscheidungen eines Entwicklers würden verschleiert.

Es bleibt festzustellen, dass die Möglichkeit, Protokollzustände über verschiedene Schnittstellen einer Komponente hinweg zu synchronisieren, die Modellierungsmöglichkeiten erheblich erweitern würde und daher wünschenswert ist. Eine solche Möglichkeit bietet etwa das SOFA Komponentenmodell [27, 33] der *Distributed Systems Research Group* der *Charles University* in Prag.

2.21.2. Mehrfache Schnittstellen-Verwendung

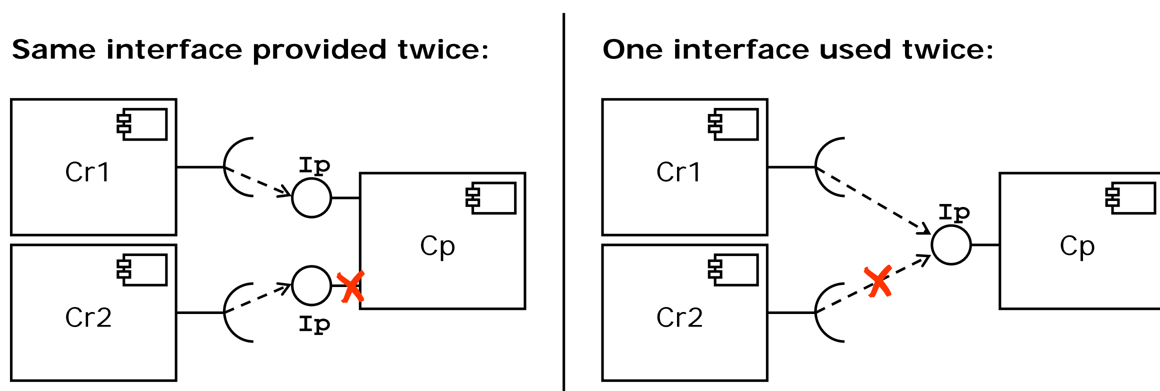


Abbildung 2.22.: Die gleiche Schnittstelle wird zweifach angeboten; eine Schnittstelle wird zweifach verwendet

Angebotene Schnittstellen Das Komponentenmodell reglementiert stark die Art und Weise, in der Schnittstellen angeboten und angebotene Schnittstellen verwendet werden dürfen. Ebenso wird die Deklaration von verwendeten Schnittstellen und der Verwendung von Schnittstellen eingeschränkt. Im Folgenden wird, getrennt nach „angeboten“ und „benötigt“, aufgezeigt, welche exakten Einschränkungen das Komponentenmodell vorgibt.

In Abbildung 2.22 werden zwei Fälle für das Anbieten von Schnittstellen unterschieden. Auf der linken Seite wird dargestellt, wie die Komponente `Cp` die Schnittstelle `Ip` zwei mal anbietet. Dieser Fall ist im Komponentenmodell nicht erlaubt. Möchten unterschiedliche Komponenten (wie im Beispiel dargestellt) die gleichen Dienste einer angebotenen Schnittstelle einer Komponente verwenden, müssen *mehrere* Komponenten die gleiche Schnittstelle anbieten.

Auf der rechten Seite wird eine alternative Modellierung dargestellt. Hier bietet die Komponente `Cp` die Schnittstelle `Ip` lediglich ein mal an. Dies ist im Komponentenmodell ein gültiges Konstrukt. Es greifen jedoch zur gleichen Zeit zwei unterschiedliche Komponenten (hier `Cr1` und `Cr2`) auf die eine Schnittstelle zu. Diese Situation ist im Komponentenmodell untersagt. Ein Assembly Konnektor kann lediglich eine *Requires*-Schnittstelle mit einer *Provides*-Schnittstelle verbinden, die ansonsten jeweils nicht über Assembly Konnektoren verbunden sind. Auch in diesem Fall lässt sich ein *Work-Around* modellieren, indem eine weitere Komponente die gleiche Schnittstelle anbietet und alle zusätzlichen Assembly Konnektoren ihre Aufrufe auf die angebotenen Schnittstellen der zusätzlichen Komponenten delegieren.

Das Kernproblem wird durch die *Work-Arounds* allerdings nicht gelöst. Es gründet in der Tatsache, dass eine Komponente je angebotener Schnittstelle nur einen einzigen Zustand besitzt (wird eine identische Schnittstelle angeboten, existiert ebenso nur ein Zustand). Erfolgen auf einer Schnittstelle, zu der ein Protokoll gehört, Aufrufe durch mehr als zwei Komponenten abwechselnd oder gleichzeitig, ist für die zugreifenden Komponenten der Protokollzustand nicht vorhersehbar. Da es keinen direkten Mechanismus gibt, mit dem der Protokollzustand einer Komponente abgerufen werden kann, können die aufrufenden Komponenten nicht mehr garantieren, einem anderem Protokoll als dem „Trivial Protokoll“ zu folgen.

Das Verbot der dargestellten Konstrukte resultiert daraus, dass für mehrfachen Zugriff auf die gleiche oder identische Schnittstelle einer Komponente die Semantik für den Protokollzustand nicht klar ist. Würde die anbietende Komponente als Singleton realisiert sein, gäbe es nur einen Komponenten-Zustand auch für mehrfach angebotene Schnittstellen – würde die Komponente einen eigenen Zustand je angebotener Schnittstelle besitzen, gäbe es damit auch einen eigenen Protokollzustand je angebotener Schnittstelle.

Insgesamt wäre eine Aufhebung dieser Einschränkung für das Komponentenmodell jedoch wünschenswert.

Übertragbarkeit auf Technologien Würde die Einschränkung, die gleiche Schnittstelle nur ein mal anbieten zu dürfen, aufgehoben, wäre die Übertragbarkeit auf reale Implementierungen von Komponentenmodell-Instanzen fraglich. Eine Komponente könnte für jede mehrfach angebotene Schnittstelle intern eine eigene Instanz vorhalten, die den internen Zustand für das jeweilige Schnittstellen-Protokoll repräsentiert. Eine Annahme des Komponentenmodells ist jedoch, dass Komponenten sich grundsätzlich nicht selbst instanziieren können.

Eine Abbildung des Komponentenmodells auf real existierende Komponentenarchitekturen (und umgekehrt) ist wichtig um beispielsweise QoS-Vorhersagen validieren zu können. Daher wird bei der Konzeption des Komponentenmodells Wert darauf gelegt, dass eine Realisierung mit existierenden Komponententechnologien (eventuell unter Zuhilfenahme von Erweiterungen) stets möglich ist.

Das in Abbildung 2.22 auf der rechten Seite dargestellte Szenario birgt zusätzlich das Problem, dass selbst bei einer Erweiterung des Komponentenmodells um eigene Zustände je identisch angebotener Schnittstelle einer Komponente in jedem Fall beide zugreifenden Komponenten auf der gleichen Instanz arbeiten. Hier wäre kein eigenes Protokoll je Schnittstelle zugreifbar.

Der Fall auf der linken Seite der Abbildung ließe sich unter Umständen mit einem

„Port“-Konzept (vgl. [39]) umsetzten. Wenn man bedenkt, dass bereits jetzt für *unterschiedliche* angebotene Schnittstellen einer Komponente angenommen wird, dass die Komponente einen eigenen internen Zustand besitzt (vgl. Kapitel 2.21.1), so ist eine Erweiterung auf *identische* Schnittstellen nicht vollständig auszuschließen. Dabei muss jedoch die Realisierung mit Hilfe realer Komponententechnologien im Hinterkopf behalten werden.

Soll im Komponentenmodell die gleiche Komponente mit der gleichen angebotenen Schnittstelle modelliert werden, lassen die konzeptionellen Begrenzungen derzeit keinen anderen Ausweg, als die gleiche Komponente mit mehreren Instanzen auf der Assembly-Ebene vorzuhalten. Damit einher geht, dass der interne Zustand der somit duplizierten Komponente sich zwangsläufig unterscheidet, sofern es keinen explizit modellierten Synchronisierungsmechanismus zwischen den duplizierten Komponenten gibt.

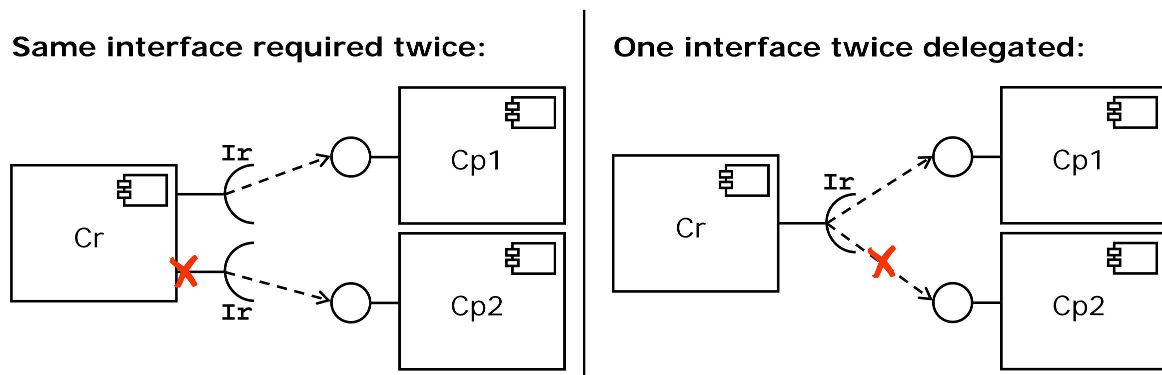


Abbildung 2.23.: Die gleiche Schnittstelle wird zweifach benötigt; eine Schnittstelle wird zweifach delegiert

Benötigte Schnittstellen Mit Analogien zu den angebotenen Schnittstellen ist es auch auf der Seite der benötigten Schnittstellen einer Komponente untersagt, dass die gleiche Schnittstelle, im Beispiel in Abbildung 2.23 links `Ir` genannt, mehr als ein mal von einer Komponente als benötigt angeben wird. Ebenso (in der Abbildung auf der rechten Seite zu sehen) darf eine benötigte Schnittstelle nicht zugleich über zwei Assembly Konnektoren mit zwei angebotenen Schnittstelle verbunden sein.

Solche Formen, eine Schnittstelle doppelt zu benötigen, ließen die Frage offen, welche der Assembly Konnektoren tatsächlich genutzt wird. Möglich wären verschiedene Strategien:

- Beide Assembly Konnektoren werden zugleich genutzt. Wie der gleichzeitige oder sequentielle Aufruf auf beiden angebotenen Schnittstellen realisiert wird, bleibt offen. Zudem bleibt ungeklärt, welche Rückgabewerte der möglichen Methodenaufrufe genutzt werden (hier lassen sich beliebig komplexe Verfahren anwenden, bspw. das Quorum- oder Consensus-Verfahren, vgl. [26], S. 265ff).

- Assembly Konnektoren werden auf der Allokationsebene auf Ressourcen (etwa LAN-Verbindungen) abgebildet. Nur bei Ausfall einer Verbindung könnte auf eine andere Ressource gewechselt werden. Hier bliebe offen, wie ein Ausfall einer Verbindung erkannt wird, wie (bei mehr als zwei redundanten Verbindungen)

die nächste Verbindung bestimmt wird und welche Verbindung initial verwendet wird.

- Würde zufällig (ohne Möglichkeit zum manuellen Wechsel auf andere Verbindungen) eine Verbindung ausgewählt, bliebe offen, wo eine Annotation von Wahrscheinlichkeiten über die Verwendung von Verbindungen hinterlegt werden könnte.

Zusammengefasst wäre ist nicht entscheidbar, welche Assembly Konnektoren, respektive Verbindungs-Ressourcen, tatsächlich genutzt würden. Damit würde das Komponentenmodell in diesem Bereich an Vorhersagbarkeit und Berechenbarkeit verlieren.

Gleichwohl gibt es Szenarien, in denen die zweite der vorgestellten Varianten wünschenswert wäre. Soll beispielsweise auf der *Requires*-Seite einer Komponente über redundante Verbindungen zu angebotenen Schnittstellen einer Komponente und durch Duplikation der anbietenden Komponenten Ausfallsicherheit erzielt werden (siehe auch *Cache Replicator Pattern* [58], S. 350ff), wird dieses nicht direkt vom Komponentenmodell unterstützt.

Zu Gunsten der Entscheidbarkeit, Vorhersagbarkeit und Berechenbarkeit verzichtet das Komponentenmodell auf die direkte Modellierbarkeit redundanter Verbindungen.

2.21.3. Rekursion von Composite Components

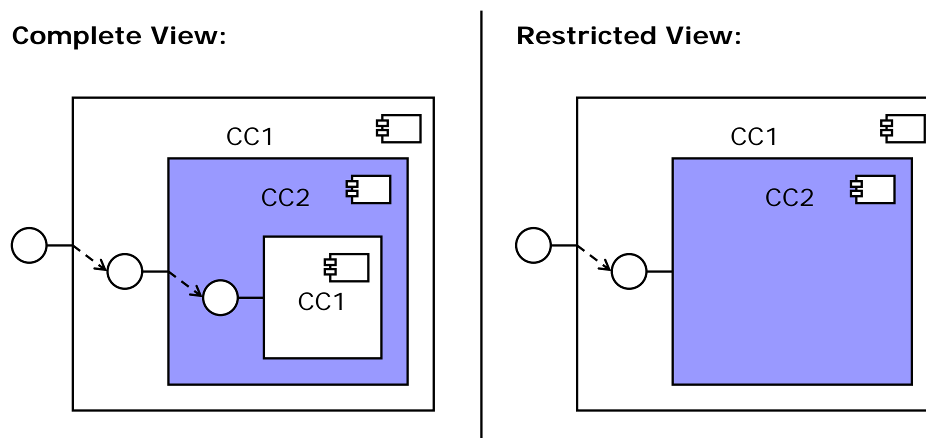


Abbildung 2.24.: Rekursive Definition einer *Composite Component* aus unterschiedlichen Sichten

Abbildung 2.24 zeigt ein Problem auf, das sich bei Komponentenmodell-Instanzen ergibt, auf die nur eine begrenzte Sicht möglich ist. Auf der linken Seite ist erkennbar, dass *Composite Component* „CC1“ zunächst „CC2“ enthält, diese jedoch wiederum „CC1“. Auf der rechten Seite wird eine eingeschränkte Sicht gezeigt, in der das Innere von „CC2“ nicht sichtbar ist. „CC1“ verwendet nach wie vor „CC2“, es ist aber keine unendliche Rekursion erkennbar.

Die skizzierte Situation kann eintreten, wenn „CC2“ als *Complete Type* modelliert wurde und unabhängig davon „CC1“ als *Implementation Type* erzeugt wird und „CC2“ intern verwendet. Der Modellierer von „CC1“ findet in der Typ-Beschreibung von „CC2“ exakt die Komponente die er benötigt und nutzt sie daraufhin. Erst in einem späteren Schritt wird „CC2“ als *Implementation Type* beschrieben. Da zu diesem

Zeitpunkt bereits der Typ von „CC1“ definiert wurde, kann der Entwickler von „CC2“ „CC1“ verwenden. Wird dabei „CC1“ bewusst als *Complete Type* betrachtet, kann der Entwickler von „CC2“ nicht erkennen, dass „CC1“ intern auf „CC2“ verweist. Auch die Verwendung von „CC2“ in „CC1“ ist einem Entwickler von „CC2“ nicht zwangläufig bekannt. Auf diese Weise entsteht eine, in diesem Fall unendliche, Rekursion, die unerwünscht ist.

Das skizzierte Szenario ist stark konstruiert und kann nur in speziellen Situationen auftreten. Insbesondere bei der *Bottom-Up*-Verwendung des Komponentenmodells, bei der Komponenteninformationen aus realen Komponenten gewonnen werden, sind Rekursionen ausgeschlossen, da hier stets ein funktionierender *Implementation Type* vorliegt. Dennoch lässt sich eine solche Rekursion auch im speziellen Fall erkennen. Setzt man voraus, dass alle Komponenten vor der Verwendung in einer Assembly vollständig mit allen inneren Komponenten als *Implementation Type* offen liegen müssen, lässt sich eine Rekursion erkennen, vermeiden oder als Fehler einer Validierung ausgeben.

(Infinite) Rekursionen von Komponenten sind im Komponentenmodell verboten.

2.21.4. Stimulus-Response-Mechanismus

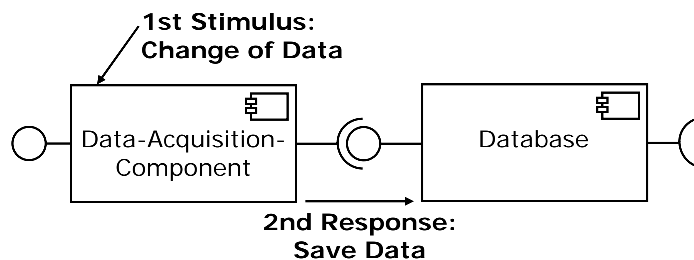


Abbildung 2.25.: Beispiel: Notwendigkeit für einen Stimulus-Response-Mechanismus

Derzeit bietet das Komponentenmodell noch keine Möglichkeit, einen Stimulus-Response-Mechanismus (vgl. [32], S. 33f) nachzubilden. Dieser würde es ermöglichen zu definieren, dass bei bestimmten Aufrufen auf dem *Provided Interface* (Stimulus) *immer* ein bestimmter Aufruf (Response) auf dem *Required Interface* erfolgen müsste.

Die Notwendigkeit, einen solchen Mechanismus nachbilden zu können, kann durch das folgende Beispiel (siehe Abbildung 2.25) illustriert werden. Eine Komponente („Data-Acquisition-Component“) soll die Änderungen ihres internen Datenzustands in einer externen Datenbank („Database“, eigene Komponente) ablegen. Damit vorgeschrieben werden könnte, dass jegliche Datenänderung in der Datenbank erfasst werden, müsste das Speichern der Datenänderungen in der Datenbank fest als Aufruf des *Required Interfaces* erfasst werden können.

Eine Erweiterung des Komponentenmodells um diesen Mechanismus ist für die Zukunft angedacht.

2.21.5. Proaktivität

Derzeit sieht das Komponentenmodell lediglich *reaktive* Komponenten vor. Stellt man sich eine größere Komponentenarchitektur vor, so könnten lediglich Aufrufe über „äußere“ Schnittstellen Aktionen im Modell anstoßen. Keine Komponente kann ohne vorausgehende Dienstaufrufe selbst Dienstaufrufe durchführen.

Zum einen ist das äußere Aufrufen von Diensten auf angebotenen Schnittstellen einer Komponentenarchitektur nicht modellierbar. Die (externen) Aufrufe können schlicht nicht erfasst werden. Dies betrifft die Zeit der Aufrufe, die Dauer der Aufrufe und die Art der aufgerufenen Dienste.

Zum anderen kann keine der Komponenten einer Architektur von sich aus Dienstaufrufe initiieren. Faktisch sind damit zunächst keine Dienstaufrufe auf Instanzen des Komponentenmodells möglich, da Initiatoren für eine Aufrufekette nicht existieren. Im Grenzbereich der Modellierbarkeit liegen Timer, die periodisch, regelmäßig und ohne zeitliche Beschränkung Dienstaufrufe durchführen. Timer ließen sich nicht aktivieren, sondern wären immer aktiv. Da Zeit derzeit nicht direkt modellierbar ist, erschwert dies zusätzlich die Funktionalität eines Timers.

Eine Erweiterung um *proaktive* Komponenten wäre für die zukünftige Erweiterung des Komponentenmodells wünschenwert.

2.21.6. Benutzerinteraktion

Benutzerinteraktion mit den angebotenen Schnittstellen einer Komponente ist nicht explizit modellierbar. Benutzer (Aktoren) können innerhalb von Komponenten (beispielsweise in einer GUI) Aktionen auslösen, die dann über die benötigten Schnittstellen der Komponente, in der der Benutzerinteraktion stattfindet, in Aufrufen auf angebotenen Schnittstellen resultieren. Ein direkter Aufruf auf angebotenen Schnittstellen ist für Aktoren nicht möglich, hätte jedoch den Vorteil, dass die Interaktion direkt – und nicht als interne Realisierung einer Komponente – modelliert werden könnte. Dazu bedürfte es etwa einer neuen Entität, die einem Aktor entspricht. Diese könnte wie eine Komponente über benötigte Schnittstellen verfügen und in fest definierbarer Weise Aufrufe auf der benötigten Schnittstelle durchführen.

Insbesondere bei der Modellierung von *Usage Profiles* würde dies den Test von Komponentenarchitekturen vereinfachen. Damit schlägt die Benutzeraktion in die gleiche Kerbe wie die in Kapitel 2.21.5 vorgestellte Proaktivität von Komponenten.

2.21.7. Dynamische Allokation, dynamischer Kontext

Die Allokation von Komponenten auf Ressourcen muss im aktuellen Verständnis des Komponentenmodells vor der Inbetriebnahme der Komponentenarchitektur erfolgen. Auf diese Weise lassen sich jedoch keine Szenarien modellieren, in denen beispielsweise zur Lastverteilung Komponenten dupliziert und dann dynamisch auf zusätzlichen Ressourcen allokiert werden – abhängig von der aktuellen Last. Damit könnten bei sehr vielen simultanen Anfragen auf den angebotenen Diensten einer Schnittstelle Performanz-Engpässe vermieden werden.

Das Komponentenmodell lässt in diesem Zusammenhang weder die dynamische Allokation zur Laufzeit noch die Verteilung der gleichen Kontext-Komponente auf mehrere Ressourcen zu.

Ebenso wäre denkbar, dass sich die Kontexte von Komponenten dynamisch ergeben. Dies wäre bei dynamischen Architekturen der Fall. Zur Laufzeit würden Konnektoren auf andere Kontext-Komponenten delegieren, womit sich der Kontext von Komponenten dynamisch änderte. Auch dieser Fall wird durch Komponentenmodell derzeit nicht abgedeckt.

2.21.8. Versionierung von Komponenten

Eine Versionierung von Komponenten ist im Komponentenmodell nicht vorgesehen. Wird eine neue Version einer Komponente geschaffen, etwa in dem die Zahl der angebotenen Schnittstellen vergrößert wird, so kann diese Version nicht mehr in Bezug zur Vorversion gebracht werden, da Versionfolgen nicht spezifizierbar sind.

Die Assembly Konnektoren des Komponentenmodells assoziieren Kontext-Rollen über ihren Identifier. Die Bindung findet daher über den Identifier statt. Wird eine neue Version einer Komponente mit einem neuen Identifier geschaffen, so besteht derzeit keine Möglichkeit die dann vorhandene neuere Version zu einer Komponente zu bestimmen. Soll die neuere Version der Komponente verwendet werden, so kann der bereits zuvor vergebene Identifier verwendet werden. Da jeder Identifier eindeutig sein muss, folgt daraus jedoch, dass die vorige Version der Komponente aus dem Komponentenmodell entfernt werden muss, damit die Eindeutigkeit der Identifier erhalten wird.

Für das Komponenten-Repository wäre eine Versionierung denkbar. Dazu müsste eine zusätzliche Assoziation zwischen Entitäten geschaffen werden, auf die die Versionsbeziehung zwischen Komponenten abgebildet werden könnte.

2.21.9. Semantikdefinition

Derzeit wird gefordert, dass untereinander erbende Schnittstellen die Semantik erhalten. Die Semantik von Diensten, die auf Schnittstellen definiert sind, lassen sich indes nicht im Komponentenmodell erfassen. Bei Diensten dient beispielsweise der Dienstname und die Signatur als Indiz für eine mögliche Semantik, liefert jedoch keine Garantien über die tatsächliche Bedeutung der Dienste.

Die Überprüfung von Semantik wäre in jedem Fall die Aufgabe von Validierungs-Algorithmen. Gleichwohl ist die Überprüfung bis dato nicht möglich, da Modellkonstrukte zur Speicherung von Semantik fehlen. Die Erfassung von Semantik über Annotationen ist derzeit nicht möglich, da zu Diensten keine Annotationen vergeben werden können. Zudem ist die Notation von Semantik weder standardisiert noch evaluiert.

2.22. Ausblick auf die Entwicklung

Das Palladio Komponentenmodell unterliegt einer permanenten Entwicklung durch die Palladio-Gruppe. Das Komponentenmodell stellt einen Gegenstand aktiver Forschung dar und variiert entsprechend. Der aktuelle Stand des Komponentenmodells wird durch ein neues Paper, das in Kürze unter [11] veröffentlicht wird, beschrieben.

Angedachte Erweiterungen des Komponentenmodells erfassen die folgenden Bereiche:

- *Runtime.* Die Laufzeitebene für Komponenten soll erfasst werden. Dazu gehört die Berücksichtigung von dynamischem Binden und das Erzeugen und Zerstören von Instanzen.

- Zustand von Komponenten. Bisher sind Komponenten nahezu zustandslos. Lediglich während der Ausführung eines konkreten SEFFs ist derzeit ein Komponentenzustand erkennbar. Außerdem lassen sich über die Protokolle der angebo-

tenen Schnittstellen Zustände der Komponenten erkennen. Ein weiter gehender Zustandsbegriff (siehe auch Kapitel 2.21) ist derzeit noch nicht möglich, soll jedoch eingeführt werden.

- Derzeit existiert keine Implementierung zum aktuellen Stand des Komponenten-Meta-Modells. Im Rahmen dieser Diplomarbeit stellt das entwickelte EMF-Modell jedoch eine (generierte) Implementierung dar.

2.23. Anmerkungen

2.23.1. Auflösen von Composite Components

Composite Components bieten lediglich eine logische Kapselung. Daher lassen sie sich vollständig auflösen, sofern ihre direkt enthaltenen (in der „contains"-Relation enthaltenen) Komponenten vollständig, einschließlich aller Delegations-Konnektoren und Assembly Konnektoren, bekannt sind. Zu diesem Zwecke zeigen Delegaten auf der Angebotsseite direkt auf die enthaltenen Komponenten, die über Delegations-Konnektoren verknüpft sind, beziehungsweise auf der Nachfrageseite direkt auf die über Delegations-Konnektoren verbundenen äußeren Komponenten.

Nicht immer können *Composite Components* jedoch aufgelöst werden. An dieser Stelle sei auf Kapitel 2.10 verwiesen. Je nach Typ-Ebene wird das Innenleben einer Komponente teils bewusst ausgeblendet und bleibt verborgen. Zudem lässt das Komponentenmodell *Composite Components* zu, die nicht ausspezifiziert sind (die interne Realisierung ist nicht vollständig bekannt). In diesen Fällen ist eine Auflösung nicht möglich.

Auch wenn sich *Composite Components* logisch auflösen lassen, so ist dies immer mit einem Informationsverlust verbunden. Da allein die Kapselung einer Menge von Komponenten eine Information darstellt, wenn man davon ausgeht, dass der Vorgang des Zusammenfassens von Komponenten nicht zufällig geschieht. Daneben ist auch der Name einer *Composite Component* bezeichnend. Zudem ist es im Komponentenmodell möglich über Annotationen Informationen zu einer zusammengesetzten Komponente zu hinterlegen (siehe auch Kapitel 2.19). Wird die Komponente aufgelöst, lassen sind auch diese Informationen nicht mehr speichern.

2.23.2. Schlechte Modellierung von Schnittstellen

Sind die Signaturlisten und Protokolle, die auf verschiedenen Schnittstelle definiert wurden, identisch, deutet dies zumeist auf ein Design-Problem von Instanzen des Komponentenmodells hin. Sind auch Zusatzattribute (z. B. Quality-of-Service-Attribute; siehe Kapitel 2.19) identisch, führt dies vermutlich zu ungewollten Inkonsistenzen zwischen einer vermeintlich identischen Schnittstelle.

3. Modellierung

3.1. Modell Driven Architecture

Der in dieser Diplomarbeit geplante Vorgehens-Prozess (Kapitel 3.3) ähnelt dem Vorgehen von Modell Driven Architecture (MDA, siehe auch [60]). MDA ist ein Standard der Object Management Group (OMG) zur Erstellung von Softwaresystemen über die Transformationen von Modellen und Programmcode. MDA wird im Allgemeinen als eine Form modellgetriebener Softwareentwicklung (MDSD, siehe [40], S. 8ff) gesehen.

MDA zeichnet sich durch modellgetriebene und generative Entwicklung von Software aus. Um eine möglichst hohe Wiederverwendbarkeit und Langlebigkeit von Modellen zu erreichen, verwendet MDA klar getrennte Abstraktionsschichten zur Modellierung von Systemen (vgl. [20]). Die Trennung von geschäftsrelevanten und technischen Informationen spiegelt sich in den Schichten (siehe [4], S. 6ff) wider:

- *Computation Independent Model* (CIM) – Natürlichsprachliche Beschreibung eines Softwaresystems. Dieser Modelltyp stellt die höchste Abstraktionsebene dar. Über das CIM wird eine Domänensicht eines zu entwickelnden Softwaresystems dargestellt.

- *Platform Independent Model* (PIM) – Fachliches Wissen, Fachlogik z. B. Geschäftslogik. Das PIM erfasst die Spezifikation der Architektur und der funktionalen Eigenschaften des zu entwickelnden Softwaresystems. Wichtig für das PIM ist die von technischen Spezifika unabhängige Beschreibung des Softwaresystems.

- *Platform Specific Model* (PSM) – Implementierungstechnologie, Services, technische Informationen. Das PSM stellt eine Erweiterung des PIMs um jene technische Aspekte dar, die die Umsetzung für eine spezifische Zielplattform erfordert.

Durch die klare Trennung von fachlicher Logik und der Technologie für die Implementierung wird eine unabhängige Wiederverwendung von fachlicher Logik beispielsweise nach einer Plattform-Migration möglich.

MDA zielt darauf, eine Brücke zwischen Modellen und Programm-Code zu schlagen. Unter der Annahme, dass abstraktere Modelle (PIM) einer geringeren Änderungshäufigkeit unterliegen, als konkrete Umsetzungen der Modelle in Quellcode und die abstrakteren Modelle zugleich kleiner sind als ihre Umsetzungen, wird der Aufwand für die Änderungen abstrakterer und kleinerer Modelle geringer ausfallen. Da die Übersetzung der abstrakten Modelle in konkrete automatisiert bewältigt werden soll, sinkt nach den Vorstellungen der MDA auch der Aufwand für komplette Software-Umsetzungen massiv.

Traditionelle Softwareentwicklung erfolgt in einer Vielzahl der Fälle händisch. Software wird entweder vollständig manuell programmiert, über Kompiler generiert oder beispielsweise unter Zuhilfenahme von GUI-Werkzeugen erzeugt. Die Modell-Erstellung

ist von der Softwareentwicklung getrennt. Die Konzepte der MDA sollen vor allem die folgenden Defizite traditioneller Softwareentwicklung ausgleichen:

- Entwicklungsaufwand / -kosten
- Konsistenz zwischen Modellen und Programm-Code über Roundtrip-Funktionalität
- Wartungsaufwand
- Interoperabilität durch Systemunabhängigkeit des PIMs

Transformationen Um inhaltlich getrennte Modell-Schichten ineinander zu überführen, definiert MDA Transformationen zwischen den Schichten (vgl. [4], S. 24).

- *PIM to PIM* erlaubt die Transformation zwischen PIMs. Man spricht in diesem Fall von Modelltransformation („Model-2-Model").
- *PIM to PSM* erlaubt die Transformation von PIMs in PSMs. Dies entspricht zumeist einer Transformation von Modell zu Code, kann aber bei einem plattformspezifischen Ziel-Modell auch von Modell zu Modell erfolgen.
- *PSM to PSM* erlaubt die Transformation von PSMs untereinander.

Transformationen bilden dabei Elemente aus dem Quellmodell auf Elemente des Zielmodells ab. Transformationen enthalten üblicherweise Erfahrungen von Softwarearchitekten und Zusatzinformationen zur Transformation und Konstruktion. Auf diese Weise können aus einfachen Modellelementen komplexere Modellelemente erzeugt werden.

MDA unterscheidet feingranular den genauen Typ von Transformationen. Als *Mapping* wird dabei die Transformation eines PIM in ein PSM für eine bestimmte Plattform verstanden (vgl. [54], S. 20 ff):

- *Model Type Mappings* spezifizieren eine Transformation von Modellen, die Typen in einer PIM-Sprache spezifizieren, in Modelle, die Typen einer PSM-Sprache verwenden. Werden Typen aus einem PIM Meta-Modell auf ein PSM Meta-Modell abgebildet, wird dies *Metamodel Mapping* genannt. Die Transformation bezieht sich in diesem Fall auf die Regeln und Algorithmen, denen alle Instanzen eines Typen des Meta-Modells folgen müssen. Das *Mapping* übersetzt die Typ-Beschreibungen in der Sprache des PIM-Modells in eine Typ-Beschreibung in der Sprache des PSM-Modells. MDA limitiert nicht auf die Verwendung von MOF zur Beschreibung von Meta-Modellen, sondern lässt beliebige andere Sprachen wie etwa CORBA IDL [59] zu.
- *Model Instance Mappings.* MDA erlaubt Transformationen, die gezielt auf einzelne Elemente eines PIMs angewendet werden sollen. Diese Transformationen können dann unter Ausnutzung der Spezifika einer Plattform in ein PSM geschehen. Dazu bedient sich MDA so genannter *Marks.* Ein *Mark* repräsentiert ein Konzept einer spezifischen Plattform (PSM) und wird auf ein PIM-Element

angewendet. Dort wird gekennzeichnet, in welcher Weise die Transformation vorgenommen werden soll.

Auch wenn *Marks* für PIM-Elemente vergeben werden, sind sie nicht Teil des PIMs, sondern plattformspezifisch. Über *Marks* können etwa Software-Architekten eine PIM-Instanz für die Transformationen zu einem PSM gezielt vorbereiten. *Marks* enthalten somit Informationen zu bestimmten PIM-Elementen, die für die Transformation genutzt werden können.

- *Combined Type and Instance Mappings* stellen eine Kombination der oben vorgestellten Formen von Transformationen dar. Würden lediglich *Model Type Mappings* angewendet, könnte ein PIM nicht um plattformspezifische Daten angereichert werden. Die Transformation würde stets in der gleichen deterministischen Weise ablaufen.

  Üblicherweise bieten sich *Model Type Mappings* zur Erhaltung von Typ-*Constraints* an, wohingegen *Marks* zur Festlegung der Spezialisierung allgemeiner Ausprägungen dienen (etwa Assoziation: „RMI navigierbar", aus [54], S. 22).

Als Sprache für „Model-2-Model"-Transformationen sieht die OMG in MDA *MOF Query/View/Transformation* (kurz MOF QVT, siehe [68]) vor.

Plattform-Begriff Der Begriff der Plattform ist nicht ganz scharf zu fassen. Je nach Abstraktionsebene wird unter Plattform etwas anderes verstanden. Ein *Enterprise Java Beans*-Entwickler unterscheidet vielleicht zwischen EJB 2.0 und EJB 3.0 als Plattform, während eine Softwarearchitekt die Wahl zwischen EJB und COM als Plattform-Entscheidung betrachtet. Selbst die Unterscheidung zwischen ECORE und UML zur Erfassung eines Meta-Modells kann als Plattform-Entscheidung verstanden werden. Dies wirkt sich entsprechend auf die Einordnung in *PIM to PIM* und *PIM to PSM* Transformationen ein.

Whitepaper Zu den Ideen und Visionen von MDA siehe auch „MDA-Whitepaper" [72].

3.2. Entwicklungswerkzeuge

Diese Diplomarbeit verfolgt mit der Modellierung des Palladio Komponenten-Meta-Modells und der Transformation der erzeugten Modelle einen MDA-Ansatz. Der Erfolg einer MDA-Entwicklung steht und fällt mit der Verfügbarkeit, dem Funktionsumfang und der Zuverlässigkeit der verwendeten Werkzeuge. Im Bereich der MDA-Werkzeuge unter Eclipse findet derzeit ein beständiges Wachstum statt. Große Software-Hersteller wie IBM und Borland lassen unter anderem Werkzeuge wie – das auch in dieser Arbeit verwendete – Graphical Modelling Framework entwickeln. Das führt dazu, dass die Zahl der verfügbaren Werkzeuge und die Fähigkeiten bestehender Werkzeuge sich stetig wandeln und weiter entwickeln.

Um über das Maximum möglicher MDA-Funktionalität zu verfügen, wurden im Rahmen dieser Diplomarbeit die Entwicklerversionen der Werkzeuge unter Eclipse verwendet. Nur diese konnten annähernd den benötigten Funktionsumfang bereit stellen.

Im Gegenzug mussten Abstriche bei der Stabilität und Reife der Werkzeuge in Kauf genommen werden. Fielen bei der Verwendung der Werkzeuge Einschränkungen im Funktionsumfang oder Fehler auf, so wird im Folgenden darauf eingegangen. Zudem wird skizziert, welche Funktionalität der verwendeten Werkzeuge benötigt wurde oder erwünscht gewesen wären.

Im Bezug auf Einschränkungen der verwendeten Werkzeuge können lediglich Aussagen für die in der Diplomarbeit verwendeten Versionen gemacht werden. Spätere Versionen der Werkzeuge können einen abweichenden Funktionsumfang und andere Fehler aufweisen.

In der Diplomarbeit wurden die folgenden Versionen der Werkzeuge verwendet. (Die Versions-Nummern der Sub-Plugins können abweichen.):

- IBM Rational Software Architect: 6.0.1.1
- Eclipse SDK: 3.2.0; Build id: I20051215-1506
- EMF (Eclipse-Plugin): 2.2.0
- EMFT (Eclipse-Plugin): None
- GMF (Eclipse-Plugin): 1.0 M5 und 1.0 M6
- GEF (Eclipse-Plugin): 3.2.0
- UML2 (Eclipse-Plugin): 1.2.0

Im Folgenden werden zahlreiche Abkürzungen für Eclipse-Technologien und -Werkzeuge verwendet. Eine kurze Erklärung dieser Werkzeuge findet sich in Kapitel A.6. Eine Einführung in die Technologien und Werkzeuge findet sich im Proposal zu dieser Diplomarbeit [44], Kapitel 1.2ff.

3.3. Entwicklungsprozess

Bereits im Proposal zu dieser Diplomarbeit (siehe [44]) wurde der in Abbildung 3.1 dargestellte Prozess zur Umsetzung der Meta-Modell-Entwicklung mit anschließender Transformation zu einem GEF-Editor vorgestellt. Dabei waren zunächst zahlreiche Variationspunkte vorgesehen, die das Risiko bei der Umsetzung minimieren sollten. Würden bestimmte Wege zur Umsetzung – aus welchen Gründen auch immer – ausfallen, könnte ein alternativer Weg verwendet werden. Im Folgenden wird der Prozess – vertikal gesehen – bis zur Mitte der Abbildung näher betrachtet (in Abbildung 3.2 vergrößert dargestellt). Der untere Teil der Abbildung, respektive des Prozess', wird in Kapitel 5 betrachtet.

Für den ersten Schritt des Prozess' war vorgesehen, das Palladio Komponenten-Meta-Modell in UML2 abzubilden. Als Werkzeuge waren Borland Together Architect 6 (siehe [14]) und IBM Rational Software Architect 6 (kurz RSA, [38]) vorgesehen. Für die Diplomarbeit lag jedoch zunächst nur eine Lizenz für das Rational-Tool vor.

Das UML2-Diagramm sollte schließlich in das Serialisierungsformat UML2 bzw. ECORE (dem Serialisierungsformat von EMF) exportiert werden, um von dort aus in EMF unter Eclipse importiert zu werden. Eine genaue Beschreibung der Transformationsprozesse findet sich in Kapitel 3.5.1.

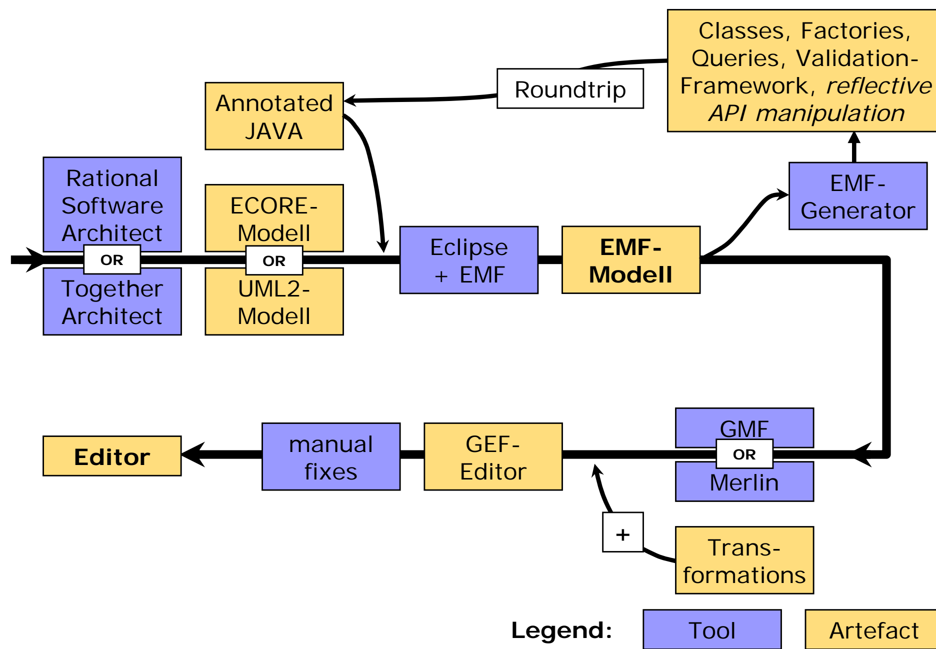


Abbildung 3.1.: Originär angestrebter Entwicklungsprozess

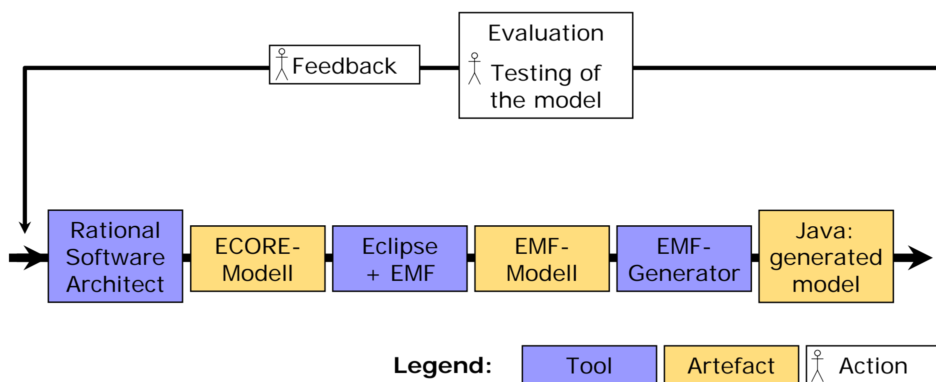


Abbildung 3.2.: Iterativer Modellierungsprozess

3.4. Anforderungen

Neben den in Kapitel 2 dargestellten Anforderungen, die sich aus den Konzepten des Komponentenmodells ergeben, werden weitere Anforderungen an die Umsetzung des Komponentenmodells im Rahmen der Diplomarbeit gestellt. Der Prozess für die Entwicklung des Komponentenmodells als EMF-Modell mit anschließender Generierung eines GEF-Editors unter Zuhilfenahme von GMF entspricht einem typischen MDA-Prozess. Die zentrale Idee zur Verwendung eines MDA-Ansatzes zur Entwicklung einer Implementierung des Komponentenmodells sowie eines GEF-Editors für das Komponentenmodell resultiert aus der hohen Frequenz von Änderungen der Konzepte des Komponentenmodells. Diese Diplomarbeit evaluiert die Umsetzung eines MDA-Prozesses von der UML-Modellierung bis zum graphischen Editor.

Da das Palladio Komponentenmodell Objekt der Forschungsarbeit der Palladio-Gruppe ist, lassen sich Änderungen weder in der Frequenz noch im Umfang vollständig abschätzen. Die Erfahrungen zeigen jedoch, dass Änderungen permanent zu erwarten sind. Um der hohen Änderungshäufigkeit begegnen zu können, wurde vom MDA-Ansatz erwartet, dass die Entkopplung über CIM, PIM, PSM und Code für die Zielplattform sowie die automatisierte Transformation und Generierung eine Verkürzung der Entwicklungszeit eines angepassten graphischen Editors nach einem Wandel des Komponentenmodells bewirkt. Zusätzlich sollte der Grad der Wiederverwendbarkeit und Langlebigkeit von Modellen nach Konzept-Änderungen möglichst hoch sein. Der Grad der Automatisierung beim Durchlauf des Prozess aus Abbildung 3.1 sollte möglich groß sein. Je höher der Anteil notwendiger manueller Eingriffe ist, desto weniger Vorteile birgt der MDA-Ansatz gegenüber einer händischen Implementierung. Zugleich darf erwartet werden, dass die Zahl möglicher Fehlerquellen durch einen hohen automatisierten Anteil am Prozess verringert wird.

Auf dem Weg von veränderten Konzepten des Komponentenmodell zu einem graphischen Editor lassen sich folgende Unteranforderungen identifizieren:

1. (UML-) Modellierung des Komponentenmodells (PIM)

   ↓ *PIM to PIM mapping*

2. EMF-Modell für das Komponentenmodell erzeugen (PIM)

   ↓ *PIM to PSM mapping*, *Type and Instance Mapping*

3. Implementierung des Komponentenmodells erzeugen (PSM)

   ↓ *PIM to PSM mapping* und *PSM to PSM mapping*

4. GEF-Editor für das erzeugte Modell generieren (PSM)

3.5. UML2-Modell

Das in diesem Kapitel beschriebene Meta-Modell des Palladio Komponentenmodells stellt *ein* mögliches Meta-Modell des Komponentenmodells dar, da bei der Meta-Modellierung stets mehrere Modellierungsalternativen zur Auswahl stehen. Bei der Frage, ob eine Assoziation über Assoziationsklasse realisiert werden soll, ist etwa zu bedenken ob zu einer Assoziation weitere Attribute definierbar sein müssen. Begründungen

für die Wahl bestimmter Modellierungsalternativen werden in diesem Kapitel explizit dargelegt.

Eine intuitive und sichere Modellierung von Instanzen des Komponenten-Meta-Modells ist dann möglich, wenn möglichst viele Konzepte des Komponenten-Meta-Modells über Modellkonstrukte und nicht über Constraints realisiert werden. Constraints liegen quer zum Meta-Modell, werden zumeist textuell dargestellt und sind daher schwerer erfassbar als Modellkonstrukte. Das Ziel der Modellierung war daher eine möglichst umfassende Umsetzung von Konzepten in den Konstrukten des Meta-Modells.

3.5.1. Transformation von UML2 zu EMF/Ecore und Java

Im Kern der Diplomarbeit sollte ein EMF-Modell des Palladio Komponenten-Meta-Modells erzeugt werden. Der oben skizzierte Prozess in der Diplomarbeit beginnt jedoch mit der Entwicklung des Meta-Modells in Form eines UML2-Diagramms. Das UML2-Diagramm wurde in einem UML2-Tool entwickelt. Um zu einem Meta-Modell in EMF-Repräsentation zu kommen, war ein Transformationsprozess notwendig.

Schritt 1. Der erste Schritt des Transformationsprozesses bestand darin, die Export-Funktion von RSA zu nutzen. Wie bereits in Abbildung 3.1 angedeutet, kamen als Export-Formate UML2 (Serialisierungsformat, nicht Diagramm-Typ) und ECORE in Frage. Da sich die in diesem Schritt verwendeten Transformationen primär auf eine andere Form der Serialisierung beziehen, soll an dieser Stelle auf den Transformationsprozess nicht näher eingegangen werden. Einschränkungen, die sich im Rahmen der Diplomarbeit ergaben, werden in Kapitel 3.5.7 behandelt.

Schritt 2. Der zweite Schritt betrifft den Import von UML2-, respektive ECORE-Modellen in EMF unter Eclipse. EMF stellt unter anderem für diese Serialisierungsformate eine Import-Funktion bereit, wobei ECORE das EMF-eigene Serialisierungsformat ist, das intern ebenfalls verwendet wird.

In diesem Schritt erfolgen kleinere Transformationen:

- Klassen der UML2-Diagramme werden in ECORE-Klassen (ECORE-Typ `EClass`) übersetzt.
- Alle Verbindungen (Assoziationen, Aggregationen und Kompositionen) der UML werden in Attribute (ECORE-Typ `Feature`) der ECORE-Klassen übersetzt, die mit den Verbindungen inzidieren. Die Navigierbarkeit der UML-Verbindungen wird dabei berücksichtigt. Ist also eine Assoziation nur in eine Richtung navigierbar (in der graphischen Notation als durchgehende Linie mit offener Pfeilspitze zu erkennen), so wird ein Attribut nur für jene ECORE-Klasse vergeben, deren Entsprechung im UML2-Diagramm eine Navigierbarkeit über die Assoziation hatte. Die ECORE-Attribute erhalten als Bezeichner (ECORE: `Name`) den in der UML vergebenen Namen und als ECORE-Typ (`EReference Type` und `EType`) den Typ der verbundenen UML-Klasse.
- Zusätzlich werden die Multiplizitäten für Attribute aus dem UML2-Diagramm übernommen. Für Assoziationen werden die annotierten Multiplizitäten verwendet, für Attribute der UML-Klassen wird `1` als Multiplizität angenommen.

- Spezialisierungen zwischen UML-Klassen werden auch in ECORE-Klassen übernommen. Das ECORE-Attribut `ESuperType` einer Klasse gibt an, welches der Name einer Oberklasse ist.

- Enumeratoren der UML werden in den ECORE-Typ `Enum` übersetzt.

- Standard-Werte der UML werden in ECORE als `Default Value` und `Default Value Literal` übersetzt.

Insgesamt entsteht auf diese Weise eine Menge von ECORE-Klassen mit Attributen. Eine Hierarchie entsteht – außer zwischen ECORE-Klasse und zugehörigen Attributen sowie eventuellen Angaben zur *Package*-Struktur, die ebenfalls aus dem UML2-Diagramm übernommen wird – nicht.

Schritt 3. Schließlich bietet EMF die Möglichkeit, zu bestehenden ECORE-Modellen Repräsentationen des Modells als Java-Quellcode zu generieren (vgl. auch Umsetzung in [41], S. 31ff).

Die Transformation folgt damit grob den folgenden Regeln:

- Jede ECORE-Klasse resultiert in einer Java-Schnittstelle sowie einer Implementierung der erzeugten Java-Schnittstelle als Java-Klasse.

- ECORE-Attribute werden zu *Gettern* und *Settern* der Java-Schnittstellen. In den dazu gehörigen implementierenden Java-Klassen werden die ECORE-Attribute zusätzlich zu den *Gettern* / *Settern* zu Instanz-Variablen.

- Entsprechend der Multiplizitäten der ECORE-Attribute werden aus den ECORE-Attributen
  - Java-`ELists` (ungetypt, vgl. „generics“ [15]) für `0..*` und `1..*` Multiplizität
  - oder Einzelwerte (*Getter* / *Setter*) der entsprechenden Java-Typen bei `0..1` und `1..1` Multiplizität.

- Die Vererbungsstruktur der ECORE-Klassen wird über die erzeugten Java-Schnittstellen abgebildet. Die Java-Schnittstellen erben in der gleichen Weise wie die ECORE-Klassen untereinander. So erbt im konkreten Fall des Palladio Komponentenmodells `Interface` von `Entity`.

- ECORE-Enumeratoren werden zu Java-Klassen, die die EMF-Klasse `AbstractEnumerator` erweitern. Die Enumeratorwerte werden zu `static final int [NAME]`, wobei `[Name]` der Bezeichner des entsprechenden ECORE-Enumeratorwerts ist.

- *Package*-Strukturen des ECORE-Modells werden in Java-*Packages* übersetzt.

Zusätzlich generiert EMF eine größere Zahl von Hilfsklassen zu jedem ECORE-Modell, darunter *Factories* (vgl. [34] S. 107ff und S. 131ff) und Validierungsklassen.

Resultat Es wurde dargestellt, weshalb eine Modellierung mit Hilfe von UML2-Diagrammen mit externen Tools prinzipiell möglich und sinnvoll erscheint. Die Transformationsprozesse arbeiten für die bis jetzt dargestellten Anforderungen fehlerfrei.

Die oben aufgeführten Schritte zeigen: Es ist möglicht mit den verwendeten MDA-Werkzeugen automatisiert aus den Vorgaben eines UML-Modells (PIM) über Transformationen zu einem PSM (Java-Quellcode) zu gelangen. Eine Anforderung von MDA ist die Unabhängigkeit von Modellen nachfolgender Transformationsschritte untereinandern. Die bis jetzt eingesetzen Modelle (UML-Diagramm/UML-Modell, ECORE und Java-Quellcode) sind untereinander vollständig unabhängig und für sich alleine verwendbar. So wäre das UML-Diagramm für weitere Transformationen nutzbar, über das ECORE-Modell ließe sich unter Verwendung modifizierter Transformationsanweisungen anderer Java-Quellcode erzeugen und der Java-Quellcode stellt ein eigenständiges, ausführbares Programm dar. Damit sind die Grundansprüche von MDA an die Modelle und durchgeführten Transformationen in diesem Bereich erfüllt.

3.5.2. Grundsätze

In Kapitel 2 wurden die Konzepte des Palladio Komponentenmodells vorgestellt. Aus diesen Anforderungen mussten, als Ausgangsbasis für den weiteren Prozess der Diplomarbeit, ein UML2-Modell sowie entsprechende UML2-Diagramme erstellt werden.

Das entwickelte UML-Modell zur Darstellung des Komponentenmodells besteht aus mehreren und zum Teil unabhängigen UML-Modellen. Eine der Anforderungen an das Modell war die Freiheit bei der Wahl von SEFF-Typen, Protokoll-Typen, der Art der Umsetzung von Identifiern und der möglichen Annotationen. Zur gleichen Zeit sollten in einem Modell verschiedenste Annotationen, SEFF- und Protokoll-Typen verwendet werden können. Endliche Automaten (FSMs) und Petrinetze stellen selbständige Modelle dar, die ebenfalls ohne das Komponentenmodell existieren können sollten. Ob endliche Automaten und Petrinetze in einer Instanz des Komponenten(-Meta-)modells als SEFFs und / oder Protokolle verwendet werden, musste grundsätzlich offen bleiben. Gleichzeitig sollten beliebige weitere SEFF- und Protokoll-Typen möglich sein.

Im Folgenden wird daher zunächst dargelegt, wie sich das Palladiokomponentenmodell aufteilt. Eine nähere Betrachtung der verwendeten Teilmodelle erfolgt in den späteren Kapiteln.

3.5.3. Sub-Modelle

Um die obig genannten Anforderungen zur Unabhängigkeit der Modelle erfüllen zu können, gibt es als zentrales Modell das Palladio Komponentenmodell („PalladioCM“) (siehe Abbildung 3.3). Um Identifier, Annotationen sowie die Spezifikation von Protokollen und SEFFs auch in anderen Projekten außerhalb des EMF-Modells verwenden zu können, sind diese jeweils in eigene Sub-Projekte ausgelagert. „PalladioCM“ hängt von diesen Sub-Modellen ab, da die Meta-Klassen dieser Sub-Modelle in „PalladioCM“ verwendet werden. Möchte ein anderes Projekt beispielsweise die gleiche Form der Identifier wie das Komponentenmodell verwenden, muss lediglich dieses Sub-Projekt importiert werden. Ein Import von „PalladioCM“ ist nicht erforderlich.

Die in Abbildung 3.3 dargestellten FSMs („FSMModel“) und Petri-Netze („PetriNetModel“) wurden im Rahmen der Diplomarbeit grob modelliert, um damit die Aus-

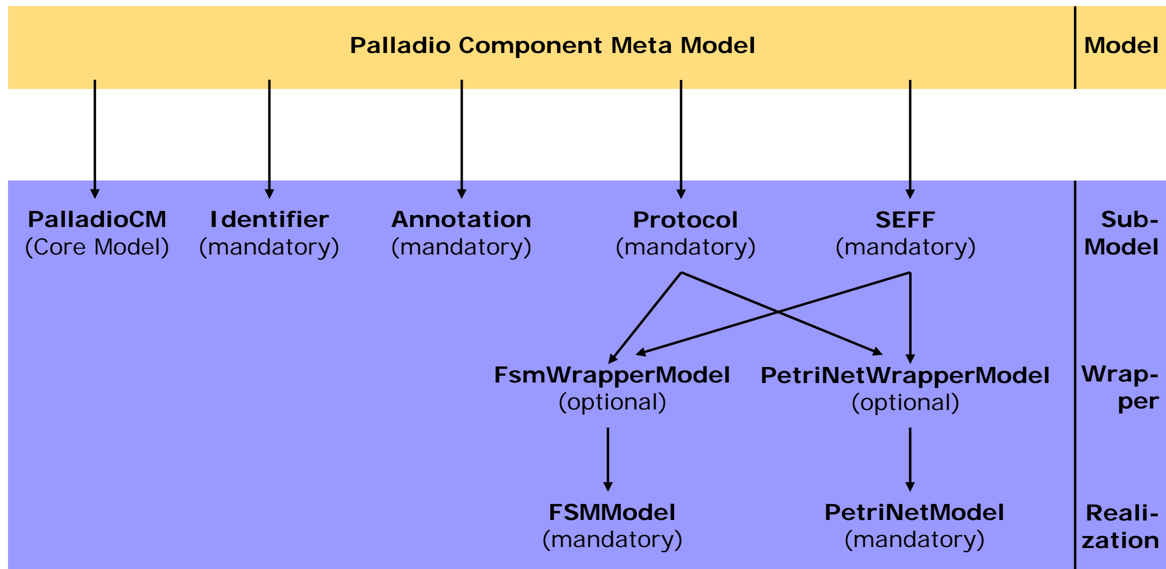


Abbildung 3.3.: Modell-Hierarchie (nicht abschließende Aufzählung von Sub-Modellen)

tauschbarkeit und Verwendbarkeit verschiedener Typen für SEFFs und Protokolle testen zu können. Diese Formen der Realisierung von Protokollen und SEFFs sind, wie im Folgenden beschrieben, über Wrapper angebunden. Da die realisierten Wrapper sowohl Protokolle als auch SEFFs implementieren, lassen sie sich für Protokolle und auch SEFFs einsetzen.

Abbildung 3.4.: Protokoll-Modell

Würden Protokolle und SEFFs lediglich über eine abstrakte Klasse realisiert, die von Realisierungen beerbt wird, führte dies dazu, dass keinerlei Struktureigeschaften vorhersehbar wären. Um ein Mindestmaß an Auswertbarkeit für Protokolle und SEFFs über vorgeschriebene Struktur-Elemente zu ermöglichen, sind für Protokolle „ServiceCall" (siehe Abbildung 3.4) und für SEFFs „ExternalServiceCall" (siehe Abbildung 3.5) als abstrakte Klassen vorgesehen. Beide Strukturelemente sollten von Realisierungen verwendet werden um Dienstaufrufe zu spezifizieren. Dazu referenzieren beide Strukturelemente eine Signatur aus „PalladioCM". Bei Protokollen können damit die Dienste spezifiziert werden, für die eine valide Sequenz bestimmt wird, bei SEFFs können externe Dienstaufrufe spezifiziert werden.

Allgemein ist als Navigationsrichtung nur die Richtung von „ServiceEffektSpecification", „Protokoll" und „ExternalServiceCall" zu Signaturen vorgesehen, um keine Abhängigkeit zwischen Instanzen des „PalladioCM"-Kerns und SEFF bzw. Protokoll-Instanzen zu erzeugen.

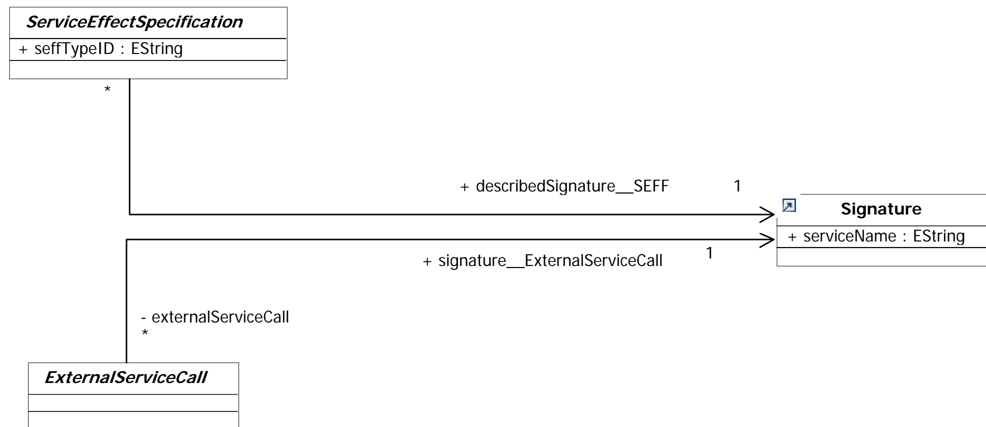


Abbildung 3.5.: SEFF-Modell

Protokoll-Modell Die abstrakte Klasse „ServiceCall“ verweist auf genau eine Signatur. Diese Signatur muss eine Signatur der Schnittstelle sein, die durch das Protokoll beschrieben wird. Wird „ServiceCall“ in konkreten Realisierungen von Protokollen zur Modellierung von Referenzen auf Dienste verwendete, lassen sich damit referenzierte Dienste eindeutig erkennen.

Zusätzliche Erweiterungen des Protokoll-Modells um weitere abstrakte Elemente wie zum Beispiel „Schritt“, „Schleife“ o. ä. sind für die Zukunft des Komponentenmodells vorgesehen.

SEFF-Modell Jeder SEFF beschreibt die Auswirkungen eines Dienstaufrufs auf der Schnittstelle eines Komponenten-Typs. Wie in Abbildung 3.5 dargestellt wird, wird der Dienst, der von einem SEFF beschriebenen wird, über eine Assoziation zwischen „ServiceEffektSpecification“ und „Signature“ dargestellt. Die externen Dienstaufrufe werden über die abstrakte Klasse „ExternalServiceCall“ modelliert und enthalten ebenfalls eine Referenz auf „Signature“.

3.5.3.1. Endliche Automaten und Petrinetze

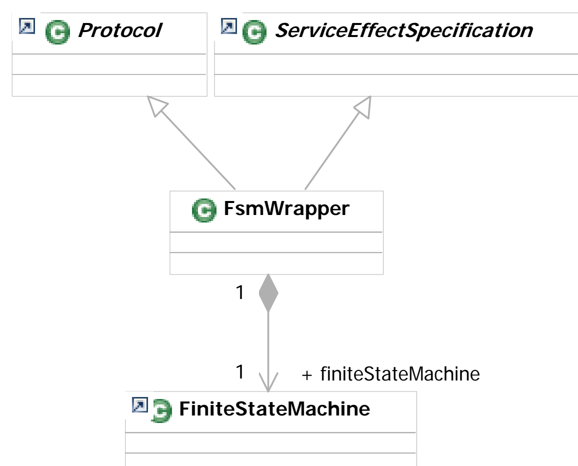


Abbildung 3.6.: FSM-Wrapper (vereinfacht)

Für FSMs und für Petrinetze gibt je ein eigenes *Wrapper*-Modell. Die *Wrapper* (vgl. auch [34], S. 171ff) stellen das Bindungsglied zwischen Komponentenmodell und FSMs („FsmWrapperModel“) sowie Komponentenmodell und Petrinetzen („PetriNetWrapperModel“) dar. Am Beispiel des FSM-*Wrappers* soll die Idee des *Wrappers* skizziert werden.

Der FSM-*Wrapper* (siehe Abbildung 3.6) enthält genau eine neue Klasse `FsmWrapper`. Das „FsmWrapperModel“ importiert über die UML2-eigene Import-Funktion das „PalladioCM“ und das „FSMModel“. Für den Test des Imports von FSMs sollten FSMs als SEFFs und Protokolle verwendet werden können. Daher erbt der `FSMWrapper` von `Protocol` und `ServiceEffectSpecification`. Ein `FSMWrapper` kann damit bereits für Protokolle und FSMs verwendet werden. Über die Komposition („composite association“ der UML2) aus genau einer `FiniteStateMachine` kapselt der `FSMWrapper` einen endlichen Automaten.

Der *Wrapper* hat Abhängigkeiten vom Protokoll-Modell, dem SEFF-Modell und dem FSM-Modell, damit erfüllt er genau die Anforderung nach Unabhängigkeit zum Komponentenmodell-Kern („PalladioCM“). Der *Wrapper* für Petrinetze ist in Analogie zum *Wrapper* für FSMs aufgebaut.

Dienstaufrufe Da die Idee Vorgaben für die Konstrukte des Komponentenmodells zu machen erst im Laufe der Diplomarbeit entstand, wurde hierfür noch keine Modellierung durchgeführt. An dieser Stelle werden daher zwei grundsätzliche Ideen zur Umsetzung skizziert.

Sollen „External Service Call“ und „Service Call“ verwendet werden, ist eine Erweiterung des skizzierten Wrappers oder eine Modifikation des FSM-Modells denkbar.

- **Abbildung.** Die erste Möglichkeit der Erweiterung des Wrappers könnte darin bestehen, eine Abbildung zwischen „External Service Call“ bzw. „Service Call“ und Transitionen eines FSMs beispielsweise in einer Tabelle abzulegen. Der Vorteil dieses Ansatzes wäre, dass das FSM-Modell komplett unberührt bleiben könnte.

- **Modifikation.** Die zweite Möglichkeit besteht in einer Modifikation von FSMs. Lässt man Transitionen beispielsweise von „External Service Call“ erben, so sind auf diese Weise externe Dienstaufrufe eindeutig in FSMs gekennzeichnet. Auf diese Weise würde jedoch das Prinzip der Trennung von FSM-Modell und Protokoll-Modell bzw. SEFF-Modell durchbrochen. Eine Abhängigkeit zum Komponentenmodell-Kern („PalladioCM“) entstünde jedoch nicht direkt, da Protokoll-Modell und SEFF-Modell eigenständige Modelle sind.

3.5.3.2. Annotationen

Für Annotationen ließen sich in der gleichen Weise *Wrapper* konstruieren wie für FSMs und Petrinetze. Da konkrete Annotationen jedoch in den meisten Fällen für sich genommen kein eigenständiges Modell darstellen, erbt das als Beispiel konstruierte „ExampleAnnotationModel“ direkt von der abstrakten Annotations-Klasse. Annotationen hängen damit direkt von „PalladioCM“ ab, zugleich bleibt das Komponentenmodell auch hier unabhängig von Realisierungen für Annotationen.

Realisierungen von Annotationen erben von der abstrakten Klassen „Annotation". Damit sind auch mehrere Realisierungen möglich, die als Datenfeld den gleichen Datentyp verwenden. Um eine unterscheidbare und eindeutige Semantik identischer Datenfeldern und -typen zu ermöglichen, wird das Attribut „annotationTypeID" der abstrakten „Annotation"-Klasse verwendet. Dieses hat als Datentyp einen Enumerator „AnnotationType". Der Enumerator dient als Speicher für alle bekannten Semantiken von Annotationen. Auf diese Weise können Annotationen über verschiedenste Modelle und Modell-Instanzen hinweg konsistent verwendet werden.

Soll eine neuer Annotations-Typ eingeführt werden, etwa ein spezieller Performanz-Wert, würde im angesprochenen Enumerator ein neuer Eintrag erzeugt, der genau diesen Performance-Wert eindeutig kennzeichnet. Immer wenn eine Annotation genau diesen Performance-Wert abbildet, wird der korrespondierende Eintrag des Enumerators als „annotationTypeID" der Annotation gesetzt.

Die Modellierung von „AnnotationType" hat jedoch zum Nachteil, dass der Enumerator zentral gepflegt werden muss, um eine einheitliche Semantik tatsächlich sicherzustellen – verteilte Modifikationen werden damit erschwert. Da das Sub-Modell für Annotationen „Annotation" aus dem Komponentenmodell ausgelagert ist, läßt sich das Modell einfach durch neuere Versionen ersetzen, die zusätzliche neue Elemente des Enumerators enthält.

Eine alternative Modellierung, bei der Vererbung von Enumeratoren ausgenutzt würde, ist seitens der UML2 nicht zugelassen – Enumeratoren können in der UML2 nicht von einander erben. Das Erben vom „AnnotationType"-Enumerator zum Zwecke der Erweiterung um weitere Annotations-Typen ist damit nicht möglich.

Die Umsetzung der Annotation wird in Abbildung A.7 verdeutlicht. Alle Entitäten (`Entity`) des UML2-Modells lassen sich mit Annotationen (Klasse `Anotation`) versehen. Da das derzeitige Komponentenmodell Annotationen genau einer Modell-Instanz exklusiv zuordnet, führen die Entitäten des UML2-Modells Annotationen als Komposition.

3.5.3.3. Standard-Datentypen für Annotationen

Die Datentypen für Annotationen werden durch das Komponentenmodell nicht beschränkt. Die Klasse *Annotation* ist eine abstrakte Klasse. Auf diese Weise sind beliebige (auch komplexe) Datentypen möglich. Diese erben lediglich von *Annotation* und können dann zur Verwendung in eine Instanz des Komponenten-Meta-Modells importiert werden.

Um die am häufigsten verwendeten Datentypen vereinfacht verwenden zu können, definiert das UML2-Modell „Annotation" ein `Package DefaultAnotationDataValue` (siehe Abbildung A.14) mit Standard-Datentypen. In diesem Paket sind die Datentypen

- `Boolean`
- `String`
- `Short`
- `Integer`
- `Long`

- `Double`

definiert. Um eine spätere Verwendung unter EMF zu ermöglichen, wurden als Datentypen die ECORE-Pendants zu den oben genannten Datentypen gewählt (also EBoolean, EString, usw.). Diese Klassen, tragen den Datenwert jeweils im Attribut `dataField`. Sollen diese Standard-Datentypen verwendet werden, können Instanzen der Klassen erzeugt werden. Zusätzlich ist lediglich das Attribut `annotationTypeID` zu setzen, um damit die Semantik des Datenwertes festzulegen.

3.5.4. Modellierung

Nachdem im vorigen Kapitel bereits die Aufteilung auf verschiedene UML2-Sub-Modelle erläutert wurde, befasst sich dieses Kapitel vor allem mit dem Kern des Palladio Komponentenmodells, im UML2-Modell auch „PalladioCM“ genannt.

Bei der Modellierung wurde darauf Wert gelegt, so viele Konzepte des Komponentenmodells wie möglich direkt und vollständig über die UML auszudrücken. Eine Basis-Konsistenz für Modell-Instanzen wird auf diese Weise durch die UML beschrieben. Um weitergehende Regeln abzubilden, wurde auf die *Object Constraint Language* (OCL, vgl. [67]) zurückgegriffen. Im Folgenden wird jedoch zunächst das reine UML-Modell beschrieben, ohne auf die *Constraints* einzugehen. Diese werden gesondert in Kapitel 3.5.5 behandelt.

3.5.4.1. Sprachwahl

Assoziieren (Assoziation), aggregieren (Aggregation) und komponieren (Komposition) bezieht sich im Folgenden auf die UML2-Notation, wie in Abbildung A.4 im Anhang dargestellt.

3.5.4.2. First Class Entities

First Class Entities (Abbildung A.16) haben im Komponentenmodell eine Sonderstellung, da diese Entitäten als Einzige ohne Abhängigkeiten zu anderen Entitäten auftreten dürfen. Die Modellierung von Instanzen des Komponenten-Meta-Modells beginnt also zunächst immer mit diesen Entitäten.

Die Klasse `CMEnvironment` stellt die zentrale Entität für alle Instanzen des Meta-Modells dar. Sie enthält den Namen und die Beschreibung zu einer Instanz des Komponentenmodells. Im in der Diplomarbeit umgesetzten Komponenten-Meta-Modell sind Instanzen nur gültig, wenn alle *First Class Entities* eine Zuordnung zu `CMEnvironment` haben. Dazu aggregiert `CMEnvironment` direkt alle *First Class Entities*, die zu einer Instanz gehören.

3.5.4.3. Factory

Das UML2-Modell musste, entsprechend dem geplanten Prozess (siehe Kapitel 3.3), anschließend in EMF unter Eclipse importiert werden können. Die mit EMF erstellten ECORE-Modelle begrenzen *Factory*-Funktionen (vgl. [34], S. 171ff) jedoch auf Entitäten, die von einer bestehenden Entität über eine *Komposition* erreicht werden können.

Um über eine zentrale erzeugende Instanz zu verfügen, wurde die Factory-Klasse realisiert. Wie Abbildung A.15 zeigt, erzeugt die Factory alle Entity-Klassen (und entsprechend alle Unterklassen). Zusätzlich kann genau ein CMEnvironment erzeugt werden. Die Attribute der Factory mit Verweisen auf CMEnvironment und Entity sind als private deklariert und damit von außen nicht zugreifbar. Da CMEnvironment eine Komponentenmodell-Instanz mit allen *First Class Entities* darstellt, würde die Factory ansonsten eine ähnliche Funktion aufweisen und damit die Trennung von Zuständigkeiten in diesem Bereich durchbrechen.

In einer alternativen Modellierung könnten Factory und CMEnvironment auch in einer Klasse realisiert werden. In diesem Fall würde jedoch der *Factory*-Mechanismus unerwünscht ein untrennbarer Teil der CMEnvironment.

Der Standard-Editor für ECORE-Modelle (vgl. Kapitel 4.2.1.3) lässt genau eine Wurzel-Klasse für Modell-Instanzen zu. Als Anforderung bestand, dass dieser Standard-Editor benutzbar sin sollte. Mit der gewählten Modellierung wird diese Wurzel-Klasse durch Factory besetzt. Alle erzeugenden Vorgänge für Instanzen von Modell-Entitäten lassen sich somit zentral abwickeln.

3.5.4.4. Identifier

Abbildung A.23 zeigt das Konzept des Identifiers. Die einzige Vorgabe, die durch das Komponentenmodell gemacht wird, ist, dass der eindeutige Schlüssel als String-Typ im Attribut id untergebracht wird. Identifier selbst ist eine abstrakte Klasse und kann damit nicht instanziiert werden. Instanziierungen geschehen im Allgemeinen über Entities (vgl. Kapitel 3.5.4.5).

Identifier sind in einem eigenständigen UML2-Modell aus dem Komponentenmodell-Kern ausgelagert. Die hierüber erzeugte Entkopplung vom Komponentenmodell-Kern ermöglicht die alleinige Verwendung von Identifiern auch in anderen Modellen. Außerdem ergeben sich Vorteile bezüglich der Implementierung unter EMF, siehe Kapitel 4.3.

Das Komponentenmodell macht keine Vorgabe, wie das id-Attribut bestimmt wird. Damit sind beliebige Realisierungen des Identifiers (bspw. über GUIDs) möglich. Von Realisierung zu Realisierung ändert sich lediglich der Prozess zur Erzeugung von Identifiern. Da die Realisierung zur Erzeugung von Identifiern im UML2-Modell nicht festgelegt ist, lassen sich in Implementierungen des Komponentenmodells beliebige Erzeugungsalgorithmen anwenden.

Vom Komponentenmodell wird gefordert (vgl. Kapitel 2.18), dass sich die zur Referenzierung genutzten Identifier in verschiedene Typen unterteilen. Damit soll ermöglicht werden, dass allein über einen Identifier Rückschlüsse auf einen Entitäts-Typ möglich sind. Referenzen auf einen falschen Entitäts-Typ ließen sich damit ausschließen. Für das in diesem Kapitel beschriebene Komponentenmodell war eine Weiterverwendung im EMF / ECORE geplant. Da ECORE Typ-Prüfungen für Referenzen auf andere Entitäten auch außerhalb des Referenzierungsmechanismus vornimmt, war eine im UML2-Modell vorgenommene Unterteilung in unterschiedliche Typen von Identifiern nicht notwendig. Konkret bedeutet dies, dass in ECORE die Typen von Entitäten nicht über den verwendeten Identifier erkannt werden, sondern über eine getrennte Typ-Prüfung. Allein dadurch wird bereits die vom Komponentenmodell geforderte Typ-Sicherheit erreicht. Auf die Einführung typsicherer Identifier konnte also verzichtet werden.

3.5.4.5. Entities

Die Definition von Entitäten wird in Abbildung A.5 visualisiert. Die abstrakte Klasse `Entity` stellt die Oberklasse aller instanziierbaren Entitäten dar. Sie definiert ebenfalls das Namens-Attribut aller Entitäten. `Entity` erbt von Identifier (externes Modell: „IdentifierModel“) und erhält damit das für Entitäten zur Referenzierung elementare `id`-Attribut. Für EMF (siehe Kapitel 4.3) hat dies den Vorteil, dass das über `Identifier` geerbte `id`-Attribut als ECORE-*ID* verwendet werden kann.

Entities sind *nicht* allesamt *First Class Entities*, auch wenn sie direkt über die *Factory* des Komponentenmodells erzeugt werden können. *First Class Entities* unterscheiden sich von den Entitäten, die von einer *Factory* direkt erzeugt werden können, darin, dass eine gültige Instanz des Komponentenmodells zwingend (je nach Konstruktion) mindestens eine *First Class Entity* voraussetzt. Das Modell führt also zu Gunsten einer einfacheren Modellierung von Modell-Instanzen eine einheitliche Behandlung von Entitäten und der Instanziierung ein.

Als Entities (Unterklassen von `Entity`) gelten Interfaces, alle Komponenten-Typen, Kontext-Komponenten, Rollen, Konnektoren und Ressourcen.

3.5.4.6. Komponenten-Typen in der Hierarchie

Das Komponentenmodell definiert insgesamt fünf Arten von Komponenten. Drei dieser Komponentenarten (`ProvidesComponentType`, `CompleteComponentType`, `ImplementationComponentType`) stellen Komponenten-Typen dar, die verbleibenden zwei Komponentenarten (`BasicComponent`, `CompositeComponent`) sind Realisierungen der Komponenten-Typen. Insgesamt wird eine Hierarchie von Komponenten gebildet. In Abbildung A.10 wird die Hierarchie dargestellt.

`ProvidesComponentType` ist die Oberklasse aller Komponenten. Davon erbt `CompleteComponentType` und davon wiederum `ImplementationComponentType`. Dadurch lassen sich beispielsweise `ImplementationComponentTypes` als `ProvidesComponentTypes` verwenden. Zusätzlich erben die Realisierungen `BasicComponent` und `CompositeComponent` ebenfalls von `ImplementationComponentType`. Die Vererbung wird in der Komponentenhierarchie jedoch primär dafür eingesetzt, Eigenschaften der Meta-Klassen für Komponenten nicht mehrfach definieren zu müssen.

Die Kernaussage der Komponentenhierarchie über die Sub-Typ- und Realisierungs-Beziehungen zwischen Komponenten-Typen ergibt sich aus den mit Stereotypen versehenen Beziehungen zwischen den Komponenten(-arten) respektive Ebenen von Komponenten. Entsprechend der Vorgaben des Komponentenmodells gibt es die Stereotypen:

- ≪`conforms`≫ zwischen `n` `CompleteComponentTypes` und `m` `ProvidesComponentTypes`.
- ≪`impl-conforms`≫ zwischen `n` `ImplementationComponentTypen` und `m` `CompleteComponentTypes`.
- ≪`realizes`≫ zwischen `n` `BasicComponents` und `m` `ImplementationComponentTypes` bzw. `n` `CompositeComponents` und `m` `ImplementationComponentTypes`.

Die Navigierbarkeit dieser Beziehungen erfolgt in allen Fällen von Sub-Typen zu Super-Typen. Auf diese Weise sind Super-Typen unabhängig von der beliebigen An-

zahl von Sub-Typen, die in einer Modell-Instanz definiert werden können – analog zur Navigierbarkeit bei Vererbungsbeziehungen etwa in C# und Java.

Im Gegensatz zu den öffentlichen Stereotypen der Beziehungen (`conforms`, `impl-conforms`, `realizes`), die jedoch keine *prüfbaren* Constraints darstellen, formalisieren OCL-Constraints die stereotypisierten Beziehungen zwischen den Komponenten-Ebenen. Zu den OCL-Constraints in der Komponentenhierarchie selbst siehe Kapitel 3.5.5.11.

Da die Beziehungen zwischen den Komponenten-Ebenen komplexer Natur sind, ließen sich die dafür geltenden Bedingungen nicht im UML2-Modell alleine abbilden. OCL-Constraints übernehmen eine weitergehende formale Definition der Bedingungen. Da die Verwendung von OCL-Constraints jedoch die Navigierbarkeit zwischen den zu überprüfenden Instanzen erfordert, mussten diese Navigationsmöglichkeiten bereits über das Meta-Modell angelegt und somit vorgeschrieben werden.

Die oben genannten Beziehungen mit Stereotypen (`conforms`, `impl-conforms`, `realizes`) dienen jedoch *nicht* dazu, dass Komponenten Listen ihrer Super-Typen, zu denen sie in Beziehung stehen, führen könnten, sondern sind eine notwendige Beziehung, die einzig zu Zwecken der Navigierbarkeit für OCL-Constraints eingeführt wurde. Aus diesem Grund sind die (UML2-)Attribute, die sich aus den Beziehungen ergeben, als `private` deklariert.

Diskussion Zu der in Abbildung A.10 dargestellten Modellierung gab es mehrere Alternativen, die im Folgenden ebenfalls dargestellt werden. Bei der letztlich gewählten Modellierung liegen alle Komponenten-Typen und die Komponenten-Realisierungen in *einem* Typen-Baum. Die unverbindliche Eigenschaft der `RequiredRoles` eines `ProvidedComponentTypes` erscheint dabei problematisch. Alle anderen Komponenten-Typen erben damit zunächst ebenfalls den unverbindlichen Charakter der `RequiredRoles`, obwohl `CompleteComponentType` und `ImplementationComponentType` ihre benötigten Schnittstellen verbindlich deklarieren müssen. Das gleiche Modellkonstrukt der `RequiredRole` ist somit in Abhängigkeit des assoziierten Komponenten-Typs einer unterschiedlichen Semantik unterworfen.

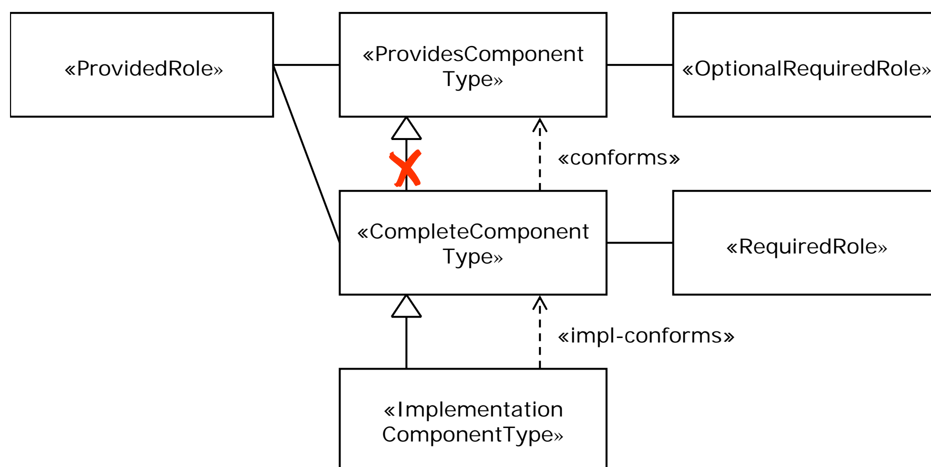


Abbildung 3.7.: Alternative zur Modellierung der Komponenten-Typ-Hierarchie

Modellierungsalternative A1 Würde man einen eigenen weiteren Rollen-Typ „OptionalRequiredRole" für den `ProvidedComponentType` einführen, der genau die unverbindliche Eigenschaft hervorhebt und die bestehende `RequiredRole` fortan mit dem `CompleteComponentType` assoziieren, müsste die Vererbung zwischen den Komponenten-Typen aufgebrochen werden. Wie in Abbildung 3.7 dargestellt wird, dürfte `CompleteComponentType` nicht mehr von `ProvidedComponentType` erben, um zu verhinden, dass „OptionalRequiredRole" von `CompleteComponentType` und `ImplementationComponentType` assoziiert werden kann.

Der Vorteil dieser Modellierungsalternative ist die klare semantische Auftrennung von verbindlichen und unverbindlichen benötigten Rollen auf verschiedene Modellkonstrukte. Nachteilig wirkt sich jedoch aus, dass Komponenten-Typen nicht mehr einheitlich behandelt werden können. Auf Grund einer fehlenden gemeinsamen Basisklasse (etwa „ComponentType") müssten `ProvidedComponentTypes` anders als `CompleteComponentTypes` und `ImplementationComponentTypes` behandelt werden. Selbst durch die Einführung einer neuen gemeinsamen Basisklasse „ComponentType", wie soeben angedacht, würde nicht jeder `CompleteComponentType` implizit seinen eigenen `ProvidedComponentType` definieren (vgl. Kapitel 2.10).

Unter den genannten Gesichtspunkten wurde diese alternative Modellierung nicht gewählt.

Modellierungsalternative A2 Im Gegensatz zur Modellierungsalternative, die im vorigen Absatz vorgestellt wurde, könnte die in Abbildung 3.7 darstellte Vererbung (mit „X" gekennzeichnet) zwischen `CompleteComponentType` und `ProvidedComponentType` erhalten bleiben. Auf diese Weise würden `CompleteComponentType` und `ImplementationComponentType` jedoch eine ererbte Assoziation zu „OptionalRequiredRole" erhalten. Da diese Komponenten-Typen ihre benötigten Rollen verbindlich deklarieren müssen, müsste eine Constraint prüfen, dass die Assoziation zu „OptionalRequiredRole" nicht verwendet wird.

Diese alternative Modellierung wäre nicht besonders intuitiv handhabbar, da je nach Komponenten-Typ die Assoziation zu „OptionalRequiredRole" nicht zugelassen wäre.

Modellierungsalternative B Wie bei der eingangs vorgestellten und letztlich durchgeführten Modellierung würde bei dieser Alternative `CompleteComponentType` von `ProvidedComponentType` und `ImplementationComponentType` von `CompleteComponentType` erben. Die `RequiredRole` würde ohne Unterscheidung zwischen verbindlichem und unverbindlichem Charakter als *ein* Modellkonstrukt existieren. `ProvidedComponentType` würde `RequiredRole` assoziieren.

Um die wechselnde Semantik einer `RequiredRole` zwischen den Komponenten-Typen explizit in einem Modellkonstrukt zu erfassen, würde ein Enumerator eingeführt, der die Werte „mandatory" und „optional" annehmen könnte. `RequiredRole` würde diesen Enumerator verwenden um anzuzeigen, welche Semantik für die benötigte Rollen gilt.

Diese Alternative hat den Nachteil, dass eine Constraint prüfen müsste, ob der Wert des Enumerators korrekt gesetzt wäre. Für den `ProvidedComponentType` müsste der Wert auf „optional", für die anderen Komponenten-Typ auf „mandatory" gesetzt werden. Damit wäre die Semantik der `RequiredRole` allein durch die Konstrukte des Meta-Modells nicht intuitiv erfassbar.

3.5.4.7. Definition von Rollen für Komponenten

Rollen stellen eine Relation zwischen Schnittstellen auf der einen Seite und Komponenten auf der anderen Seite dar. Abbildung A.22 verdeutlicht die Relation für den Fall der `ProvidedRole` und der `RequiredRole`. `ProvidedRole` und `RequiredRole` sind – wie oben diskutiert – für alle Komponenten-Typen definiert. Daher sind die entsprechenden Relationen auf `ProvidesComponentType` definiert. Da alle Komponenten von `ProvidesComponentType` erben, können auch alle anderen Komponenten-Typen `ProvidedRoles` assoziieren.

Rollen (egal ob *Provided* oder *Required*) assoziieren genau eine Schnittstelle und eine Komponente.

3.5.4.8. Vererbungsstruktur von Rollen

Um Rollen gemeinsame Attribute / Eigenschaften geben zu können, existiert mit `Role` eine gemeinsame abstrakte Basisklasse für `ProvidedRole` und `RequiredRole`. In der aktuellen Ausprägung des Komponentenmodells werden jedoch keine Attribute / Eigenschaften auf Role definiert. Die Vererbungsstruktur wird in Abbildung A.21 dargestellt.

3.5.4.9. Schnittstelle

Eine Schnittstelle (Klasse `Interface`) setzt sich im UML2-Modell aus `0..*` Protokollen sowie `0..*` Signaturen zusammen. Da Protokolle und Signaturen abhängig von einer Schnittstelle sind, ist die Beziehung zu Protokollen und Signaturen als Komposition realisiert. Abbildung A.17 verdeutlicht den Aufbau von Schnittstellen.

Die Vererbungsbeziehungen von Schnittstellen in Instanzen des Komponenten-Meta-Modells erfolgt über die mit `extends` stereotypisierte Relation. Da nicht jede Schnittstelle eine andere erweitern muss, ist die Multiplizität für `parentInterface` im Modell mit `0..*` angegeben.

Um für OCL-Constraints Rekursionen zu erlauben, war es notwendig, auch bei Schnittstellen eine Hilfsbeziehung einzuführen. `ancestorInterfaces` ist die über die transitive Eigenschaft von `extends` ermittelte Menge aller (indirekt) geerbten Ahnen-Schnittstellen. `ancestorInterfaces` ist ein `private`-Attribut, da es lediglich für OCL-Constraints benötigt wird (siehe hierzu Kapitel 3.5.5.11).

3.5.4.10. Signaturen

Signaturen, wie sie in Abbildung A.17 dargestellt werden, bestehen auch im UML-Modell aus Rückgabetyp (`returntype`), einer geordneten Menge von Parametern (`parameters`) und einer ungeordneten Menge von Exceptions (`exceptions`). Da Rational Software Architect für Attribute nicht zulässt Modifizierer wie `bag` oder `ordered` zu vergeben, kann im UML2-Modell nicht kenntlich gemacht werden, dass Exceptions eine ungeordnete und Parameter eine geordnete Menge sind. Die UML2 definiert sehr wohl solche Modifizierer, siehe hierzu [69], S. 82ff. In Ermangelung weitergehender Unterstützung durch die Modellierung-Software musste auf die zusätzliche Charakterisierung der Beziehung verzichtet werden.

Um innerhalb von Signaturen direkt Datentypen, Parameter und Exceptions erzeugen zu können, wurde für die Modellierung der Beziehung zwischen den genannten Klassen eine Komposition gewählt.

Der Rückgabetyp von Signaturen ist über die Klasse `DataType` realisiert. `DataType` selbst stellt zunächst einen beliebigen Datentyp dar. Auf `DataType` kann eine eigenständige Vererbungsstruktur aufgebaut werden, indem die `extends`-Relation verwendet wird. Ein `DataType` kann hierüber von `0..* DataTypes` erben.

Derzeit erfolgt explizit keine Nutzung von Komponenten-Schnittstellen als `DataType`. Auf diese Weise wird vermieden, dass modelliert werden kann, dass Komponenten bei externen Dienstaufrufen unter Ausnutzung von Polymorphie Komponenten-Instanzen als Parameter übergeben können. Zu Gunsten der Erhaltung der statischen Bindung von Kontext-Komponenten untereinander ist damit vorhersehbar, auf welchen Komponenten-Typen tatsächliche Dienstaufrufe durchgeführt werden. Änderungen dieser Modellierung von `DataType` sind für zukünftige Versionen des Komponentenmodells möglich.

Parameter (gleichnamige Klasse im UML2-Modell) führen einen Namen und liegen geordnet zu einer Signatur vor. Wie bereis angesprochen wurde, trifft hier eine Einschränkung von RSA zu: das zur expliziten Kennzeichnung von geordnet vorliegenden Mengen gedachte Zusatzattribut `ordered` kann für `parameters` nicht vergeben werden. Jeder Parameter muss einen Modifizierer verwenden. Um eine Auswahl aus einer bestehenden Menge von Modifizierern zu ermöglichen, gibt es den Enumerator `ParameterModifier`.

Als Datentyp für Parameter wird gleichfalls auf `DataType` zurückgegriffen werden. Auf diese Weise können die gleichen Datentypen – einschließlich Vererbungsstruktur – verwendet werden.

Exceptions (Klasse `ExceptionType`) enthalten neben einem Namen eine Ausnahmenachricht, die den Fehler beschreibt.

3.5.4.11. Kontext

Komponenten Der Kontext (siehe Abbildung A.13) wird im UML2-Modell mehrschichtig reflektiert, zunächst einmal in Form der Klasse `ContextComponent`. Diese stellt einen Komponenten-Typ in einem Kontext („einer Verwendung“) dar. Da Kontext-Komponenten zu jedem Komponenten-Typ in der Komponenten-Hierarchie auftreten können, definiert das UML2-Modell Kontext-Komponenten für den `ProvidesComponentType`. Durch die Vererbungsstruktur innerhalb der Komponenten-Typen können auch alle anderen Komponenten-Typen in einem Kontext auftreten. Kontext-Komponenten unterscheiden sich bezüglich der im Meta-Modell vergebenen Attribute nicht untereinander, auch wenn sie einem anderen Komponenten-Typ zugeordnet sind.

Um Kontext-Komponenten in EMF-Modell-Instanzen identifizieren zu können, tragen sie zusätzlich das `private` Attribute `description`. Insbesondere mit dem durch EMF generierbaren Standard-ECORE-Editor ließen sich Kontext-Komponenten ansonsten nicht unterscheiden. Das `description`-Attribut ist ein Hilfsattribut ohne weitere Bedeutung für das Komponentenmodell.

Die konkrete Zuordnung von Kontext-Komponenten zu einem Kontext erfolgt über die Klasse `Context`, mit der jede Kontext-Komponente assoziiert wird. Dazu gibt es eine `1:1` Beziehung zwischen `ContextComponent` und `Context`. `Context` enthält wiederum zur einfacheren Identifikation ein privates `description`-Attribut.

Die Navigation zwischen `ProvidesComponentType` und `ContextComponent` ist lediglich von Kontext-Komponente zu Komponenten-Typ erlaubt. Dadurch wird erreicht, dass Komponenten-Typen, die in einem Repository liegen, keine Abhängigkeit zu ihrer Verwendung im Kontext (`0..*` mal) erhalten. Anders herum sind Kontext-Komponenten keine Duplikate von Komponenten-Typen im Kontext, duplizieren also keine Attribute des Kompontenten-Typs, sondern referenzieren ihren Komponenten-Typ.

Rollen Neben den Kontext-Komponenten liegen auch Kontext-Rollen in einem Kontext vor. Für die *Requires*- und *Provides*-Role sind die Klassen identisch aufgebaut. Es gibt zu jeder Rolle (im Folgenden jeweils für *Provides* und *Requires*) `0..*` Kontext-Rollen. Auch Kontext-Rollen enthalten zur einfacheren Identifikation ein privates `description`-Attribut. Darüber hinaus sind Kontext-Rollen genau einem Kontext zugeordnet. Dieser wird im UML2-Modell über die bereits angesprochene Klasse `Context` realisiert.

Auch die Navigation von Rollen zu Kontext-Rollen ist, wie bereits bei Komponenten-Typen und Kontext-Komponenten, nicht möglich. Damit sind die im Repository definierten Rollen unabhängig von ihrer Verwendung im Kontext. Kontext-Rollen referenzieren dagegen Rollen aus dem Repository.

Anwendung des Kontext-Konzepts Insgesamt gibt das UML2-Modell damit die Möglichkeit `1..1` Kontext-Komponenten und `0..*` Kontext-Rollen einem eindeutigen Kontext zuzuordnen. Über die dargestellten Modell-Konstrukte kann indes nicht garantiert werden, dass jede zu einem Komponenten-Typen deklarierte Rolle eine Entsprechung im Kontext besitzt. Das heißt, dass es Kontext-Komponenten geben kann, für die weniger Kontext-Rollen oder Kontext-Rollen anderer Komponenten-Typen im Kontext der Kontext-Komponente existieren. Die Konsistenz zwischen den Rollen eines Komponenten-Typs und den Kontext-Rollen einer Kontext-Komponenten kann erst über Constraints geprüft werden.

Der Kontext wird bei dieser Art der Modellierung nicht als Zusatzattribut von Komponenten-Typen gesehen, sondern als immanente Eigenschaft, die „Kontext-Entitäten“, namentlich Kontext-Komponenten, *Provides* Kontext-Rollen und *Requires* Kontext-Rollen, besitzen. Auf diese Weise wird klar zwischen der Typ-Beschreibung einer Komponente und der Verwendung einer Komponente im Kontext getrennt, sowie zwischen den Rollen, die ein Komponenten-Typ anbietet und benötigt (also der Typ-Definition dienen) und den Kontext-Rollen, die eine Verwendung von Rollen ermöglichen. Wie Kapitel 3.5.4.12 zeigen wird, kann eine *Verwendung* von Rollen nur über Kontext-Rollen erfolgen.

3.5.4.12. Assembly Konnektoren

Rollen Assembly Konnektoren (Klasse `AssemblyConnector`, siehe Abbildung A.8) verbinden zwei Rollen miteinander. Wie bereits im vorangehenden Kapitel angedeutet wurde, müssen Assembly Konnektoren *Kontext*-Rollen verbinden – eine angebotene Kontext-Rolle und eine benötigte Kontext-Rolle.

Komponenten-Typen (diese liegen in einem Repository) können beliebig oft verwendet werden (in einem Kontext). Würden Assembly Konnektoren zwei Rollen eines

Komponenten-*Typs* assoziieren, wäre die Semantik keineswegs klar. Konsequenter Weise wären die Rollen aller Verwendungen eines Komponenten-Types über den definierten Assembly Konnektor verbunden. Genau dieses Verhalten ist jedoch nicht gewünscht, da sich unter Annahme eines bestehenden Repositories, in dem Assembly Konnektoren existierten, keine individuellen Komponentenarchitekturen (Komponentenverwendung) mehr erstellen ließen. Die Architekturen wären durch das Repository determiniert.

Daher wurden im UML2-Modell Assembly Konnektoren auf Kontext-Rollen definiert. Über den Kontext ist der Verwendungsort einer Komponente und ihrer Rollen eindeutig, womit auch eindeutig ist, welche Kontext-Rollen über einen Assembly Konnektor verbunden werden.

Repository Zugleich wird die Idee des Komponenten-Typ Repositories gestützt. Im Repository werden lediglich Komponenten-Typen definiert. Informationen zur Struktur der externen Verbindung von Komponenten sind nicht Teil des Repositories, sondern erst über Kontexte möglich.

Das Repository für Komponenten-Typen wird im UML2-Modell nicht explizit modelliert. Vielmehr sind alle, in Instanzen des Komponenten-Meta-Modells definierten, Komponenten-Typen automatisch Teil des Repositories. Im Gegensatz dazu sind alle Kontext-Komponenten und -Rollen als Teil der Komponentenarchitektur zu verstehen.

3.5.4.13. Composite Component

Zusammengesetzte Komponenten (Klasse `CompositeComponent`, Abbildung A.11) aggregieren vier Klassen des UML2-Modells:

- `0..*` Assembly Konnektoren, wie bereits oben angedeutet
- `0..*` Kontext Komponenten
- `0..*` `RequiredDelegationConnectors`
- `0..*` `ProvidedDelegationConnectors`

sowie die über ihre Eigenschaft als Unterklasse des `ImplementationComponentType` bestehenden Attribute. Die „Beinhalten-Relation“ zwischen einer *Composite Component* und den oben genannten Klassen wird über den `contains` Stereotyp der Aggregation ausgedrückt.

Delegation-Konnektoren Die beiden letzten Klassen der vorangehenden Aufzählung wurden als UML2-Modell-Konstrukt noch nicht eingeführt. *Required* Delegations-Konnektor und *Provided* Delegations-Konnektor *müssen* in einer *Composite Component* erfasst werden. Beide sind – ähnlich den Rollen – analog zu einander aufgebaut (siehe Abbildungen A.18, A.19).

Delegations-Konnektoren aggregieren genau eine innere und genau eine äußere Kontext-Rolle einer *Composite Component* – entweder auf der angebotenen oder benötigten Seite. Damit können Sie Verbindungen zwischen inneren und äußeren Kontext-Rollen abbilden.

Assembly Konnektoren und Kontext Komponenten Assembly Konnektoren können exklusiv nur auf der Assembly-Ebene oder (im Sinne von *exklusiv-oder*) in *Composite Components* vorkommen. Daher wurde die Multiplizität für die Aggregation von `parentCompositeComponent` als `0..1` gewählt.

Kontext-Komponenten, die von einer *Composite Component* erfasst werden, sind, entsprechend des Stereotyps der Aggregation, innere Komponenten einer *Composite Component*.

3.5.4.14. Konnektoren

Bis jetzt wurden insgesamt drei Typen von Konnektoren vorgestellt. In Abbildung A.12 werden diese, um Super-Klassen erweitert, in einer Hierarchie dargestellt. Um gemeinsame Attribute für Delegations-Konnektoren anlegen zu können, wurde die abstrakte Super-Klasse `DelegationConnector` eingeführt. Zusammen mit `AssemblyConnector` erbt `DelegationConnector` von `Connector`. Diese abstrakte Super-Klasse kann gemeinsame Attribute aller Konnektoren erfassen.

In der derzeitigen Ausprägung des UML2-Modells enthalten weder `Connector` noch `DelegationConnector` zusätzliche Attribute. Damit dienen diese abstrakten Super-Klassen zukünftigen Erweiterungen des Komponentenmodells.

3.5.4.15. Ressourcen

Die Unterteilung des Komponentenmodells auf zwei Arten von Ressourcen findet ihre direkte Entsprechung im UML2-Modell. Wie Abbildung A.20 zeigt, gibt es zwei Klassen zur Abbildung von Ressourcen:

- `CalculatingResource` stellt Ressourcen dar, die berechnender Natur sind.
- `NonCalculatingResource` stellt Ressourcen dar, die keine allgemeinen Berechnungen durchführen können. Hierunter fielen beispielsweise LAN-Verbindungen.

Um Attribute vergeben zu können, die beiden Ressourcen-Typen gemein sind, wurde die abstrakte Super-Klasse `Resource` für Ressourcen eingeführt. Hier können entsprechende Attribute definiert werden. Derzeit sind in dieser Super-Klasse keine gemeinsamen Attribute hinterlegt.

3.5.4.16. Allokation und Ressourcen

Allokation Die Modellierung der Allokation, also die Abbildung von Komponenten auf Ressourcen, wird in Abbildung A.6 beschrieben. Die Allokation erfolgt grundsätzlich für kontext-bezogene Entitäten des Komponentenmodells. So ist eine Allokation von Komponenten-Typen (aus dem Repository) nicht erlaubt und im UML2-Modell nicht möglich.

Kontext-Komponenten werden, wenn sie allokiert werden, auf genau einer berechnenden Ressource allokiert. Der hierzu definierte Stereotyp lautet `deployed-on`.

Ebenfalls können Assembly Konnektoren allokiert werden. Diese haben vordergründig keinen Kontext-Bezug. Da Assembly Konnektoren jedoch auf Kontext-Rollen definiert sind, lässt sich der Kontext transitiv ermitteln. Assembly Konnektoren werden, wenn sie allokiert werden, auf nicht-berechnenden Ressourcen allokiert. Auch hier lautet der Stereotyp `deployed-on` für die allokierende Assoziation.

Ressourcen-Architektur Berechnende Ressourcen sind untereinander in einer eigenständigen Architektur über nicht-berechnende Ressourcen verbunden. Zwar funktionieren nicht-berechnende Ressourcen entsprechend der Konzepte des Komponentenmodells bi-direktional; um im UML2-Modell jedoch Anfangspunkt und Endpunkt einer Verbindung unterscheiden zu können, wurden die zwei zwischen berechnender Ressource und nicht-berechnender Ressource bestehenden Assoziation mit *Source* und *Target* bezeichnet.

Über nicht-berechnende Ressourcen lassen sich damit paarweise zwei berechnende Ressourcen mit einander verbinden. Damit lassen sich beliebige Verbindungsstrukturen durch das UML2-Modell abbilden, sofern genau zwei berechnende Ressourcen mit einander verbunden werden sollen. Andere Verbindungsstrukturen – etwa ein Netzwerk-Switch ([78]) – die einer Sternstruktur entsprechen, lassen sich dennoch abbilden. Dazu wird die verteilende Komponente (Netzwerk-Switch) über eine berechnende Ressource nachgebildet. Damit sind beliebig viele Verbindungen zu anderen berechnenden Ressourcen modellierbar. Tatsächlich erscheint diese Modellierung angebracht, da – um im Beispiel zu bleiben – Netzwerk-Switches berechnende Funktionalität besitzen. Sollten solche Modell-Instanzen mit Sternstruktur modelliert werden, müsste jedoch bedacht werden, dass Netzwerk-Switches üblicherweise keine Komponenten-Allokation (etwas für COM- oder EJB-Komponenten [50, 73]) erlauben.

In diesem Bereich sind daher möglicherweise Änderungen für zukünftige Versionen des Komponentenmodells (auch in seiner UML2-Repräsentation) vorzusehen.

3.5.4.17. Service Effect Specification

Service Effect Specifications werden exklusiv je *Basic Component* definiert. Daher komponiert sich eine `BasicComponent` aus der abstrakten Klasse `ServiceEffectSpecification`, siehe auch Abbildung A.9. Die Verwendung einer abstrakten Klasse für SEFFs entspricht dem Grundsatz des Komponentenmodells, keine spezielle Form für SEFFs vorzugeben.

Zu einer *Basic Component* sind `0..*` SEFFs möglich. Das UML2-Modell folgt damit der Vorgabe des Komponentenmodells, dass parallel zu einer *Basic Component* SEFFs in verschiedenen Formen vorliegen können. Um überprüfen zu können, ob tatsächlich keine SEFF-Typen doppelt für eine *Basic Component* verwendet werden, müssen SEFFs im Attribut `seffTypeID` in eindeutiger Weise den Typ des SEFFs angeben.

3.5.5. OCL-Constraints

In diesem Kapitel werden die im UML2-Modell verwendeten OCL-Constraints erläutert, die im Rahmen der Diplomarbeit verwendet wurden, um das Komponenten-Meta-Modell inklusive ihrer Konsistenzbedingungen zu definieren. Zunächst wird eine kurze Einführung in OCL gegeben. Anschließend werden die OCL-Bedingungen des UML2-Modells behandelt. Hier wird ein besonderes Augenmerk auf Probleme und Einschränkungen bei der Verwendung von OCL gelegt. Zudem werden die Ideen der einzelnen OCL-Constraints aufgezeigt, da sich diese üblicherweise nicht direkt durch Lesen des OCL-Codes ergeben.

3.5.5.1. Einführung

Invarianten OCL – die *Object Constraint Language* der OMG [67] – wurde im Rahmen der Erstellung des UML2-Modells des Komponentenmodells eingesetzt, um Konsistenzbedingungen (Invarianten) auszudrücken, die sich nicht mit den Modellkonstrukten der UML2 abbilden ließen.

OCL ist eine drei-wertige Kleene-Logik (vgl. [5], S. 25ff) – inklusive der Definition von Gleichheit – um Constraints für Modell-Instanzen zu definieren, deren Struktur durch ein UML-Modell beschrieben wird.

Eine OCL-Constraint kann mit RSA nur für genau eine Klasse eines UML2-Modells verwendet werden – diese Klasse stellt den Kontext einer OCL-Constraint dar. Aus einer OCL-Constraint wird ein boolscher Wert berechnet. Eine Modell-Instanz ist gültig, wenn alle im Meta-Modell spezifizierten Constraints für die Entitäten der Instanz gültig sind. Mit Hilfe von OCL-Constraints kann man über die Konstrukte von MOF und damit schließlich über alle Attribute und Assoziationen eines UML2-Modell navigieren und damit Bedingungen für alle Entitäten definieren, die von einem initialen OCL-Kontext aus erreichbar sind. OCL-Constraints beziehen sich zumeist auf Mengen von Instanzen von Modellentitäten. Daher bietet OCL zahlreiche Ausdrücke um Projektionen, Vereinigungen, Schnitt- oder sonstige Mengen-Operationen auf Mengen durchzuführen. Zusätzlich gibt es Schleifen- und `if then else`-Konstrukte, um Bedingungen zu definieren.

OCL kennt einen Satz von Basis-Datentypen und Methoden auf diesen Datentypen. Selbst definierte Variablen und Methoden lassen sich je OCL-Ausdruck angeben. Für Mengen bietet OCL ebenfalls vordefinierte Datentypen an, die zugleich die oben genannten grundlegenden Mengen-Operatoren unterstützen.

OCL ist grundsätzlich eine getypte Sprache, die ebenfalls Konstrukte wie explizite und implizite *Type-Casts* umfasst.

Vor- und Nachbedingungen Neben Invarianten lassen sich über OCL Vor- und Nachbedingungen von Operationen und Methoden definieren. Damit ist es beispielsweise möglich, Bedingungen zu definieren, die vor der Ausführung einer Methode, und jenen Bedingungen, die nach der Ausführung einer Methoden gelten müssen. Vor- und Nachbedingungen wurden im Rahmen der Diplomarbeit für das Komponentenmodell nicht benötigt.

Query-Sprache Wie bereits beschrieben, erlaubt OCL eine Navigation und Abfrage auf MOF-Modellen. Auf diese Weise lassen sich Anfragen beschreiben, die gänzlich unabhängig von einer konkreten Programmiersprache sind. Im Rahmen der Diplomarbeit wurde OCL nicht ausschließlich als *Query*-Sprache verwendet. Vielmehr waren *Queries* notwendig, um im UML2-Modell zu navigieren.

Allgemeines OCL ist eine reine Spezifikationssprache. Das heißt, dass alle OCL-Constraint frei von Seiten-Effekten sind.

Da die Namen nicht erfüllter OCL-Constraints bei der Modellierung von Instanzen des Komponenten-Meta-Modells häufig als Teil einer Fehlermeldung angezeigt werden, wurden möglichst aussagekräftige Namen gewählt. Aus diesem Grund sind die Namen einiger der im Folgenden beschriebenen Constraints sehr lang.

Die verwendeten OCL-Constraints sind bildlich in den UML2-Abbildungen im Anhang (Kapitel A.4) dieser Diplomarbeit erfasst. Leider unterstützt RSA keine linksbündige graphische Darstellung der Constraints für die Druck-Ausgabe. Komfortabel sollten sich die Constraints der mit dieser Diplomarbeit veröffentlichten CD-ROM entnehmen lassen. Dort findet sich zum einen eine HTML-Darstellung, aus dem „Web-Report“ von RSA, die die Constraints als Text enthält. Zum anderen ist dort das Serialisierungsformat (`.emx`) von RSA zu finden, das das gesamte Komponentenmodell enthält.

In den nächsten Kapiteln werden die Bezeichnungen der Constraints wie folgt angegeben:

[Kontext]::[Constraint-Name] Durch den Kontext und den Namen der Constraint ist der Bezug eindeutig. Der vollständige Kontext lautet stets PalladioCM::[NameDer-Entität]::[ConstraintName], wird jedoch aus Platzgründen nur verkürzt wiedergegeben.

3.5.5.2. Kontext-Rollen

ContextProvidedRole::OnlyUseByAssemblyConnectorXORDelegationConnector
Kontext-Rollen, die angeboten werden, dürfen nur mit genau einem oder keinem Konnektor verbunden sein, wie bereits in Kapitel 2.21.2 diskutiert wurde, würden sich im aktuellen Komponentenmodell ansonsten Semantikprobleme bezüglich des internen Zustands der über die Rolle verbundenen Komponente ergeben.

Das UML2-Modell erlaubt für angebotene Kontext-Rollen jedoch `0..1` Verbindungen zu Delegations-Konnektoren und `0..1` Verbindungen zu Assembly Konnektoren. Damit wären im schlimmsten Fall eine Verbindung zum Delegations-Konnektor und auch eine Verbindung zum Assembly Konnektor möglich. Diese Constraint prüft, dass nur `0..1` Verbindungen zu einem Konnektor (allgemein) existieren.

Diese Bedingung konnte nicht in UML2 ausgedrückt werden, da hierfür angebotene Kontext-Rollen zu der existieren Oberklasse `Connector` eine auf `0..1` beschränkte Assoziation haben müsste – ohne, dass weitere Assoziationen zum Assembly Konnektor und Delegations-Konnektor bestünden. Über die allgemeine `Connector`-Klasse ließen sich jedoch beispielsweise angebotene und benötigte Delegations-Konnektoren nicht mehr unterscheiden, so dass in diesem Bereich zusätzliche Constraints erforderlich gewesen wären, um das Komponentenmodell korrekt abzubilden.

ContextRequiredRole::OnlyUseByAssemblyConnectorXORDelegationConnector
Benötigte Kontext-Rollen müssen, vollkommen analog zu den angebotenen Kontext-Rollen, der gleichen Constraint genügen.

3.5.5.3. Assembly-Konnektor

AssemblyConnector::contextIDsOfProvidesRoleNotEqualRequiresRole
Diese Constraint stellt sicher, dass die Identifier des Kontexts der angebotenen Kontext-Rolle eines Assembly Konnektors verschieden vom Identifier des Kontexts der benötigten Kontext-Rolle sind. Ansonsten würde eine Komponenten-Instanz Aufrufe auf sich selbst durchführen. Ein solches Verhalten dürfte nicht beabsichtigt sein, da Aufrufe auf der eigenen Instanz nicht über eine benötigte Schnittstelle erfolgen sollten, sondern intern in einer Komponente.

3.5.5.4. Basic Component

BasicComponent::noSeffTypeUsedTwice
Das Komponentenmodell erlaubt, dass eine Basic Component zur gleichen Zeit durch mehrere SEFFs beschrieben wird, fordert jedoch, dass der gleiche SEFF-Typ nicht mehrfach verwendet wird. Diese Constraint sorgt für die Einhaltung der Bedingung des Komponentenmodells.

3.5.5.5. First Class Entities

Die in diesem Kapitel aufgeführten Constraints ähneln sich stark. `CMEnvironment` hält in Listen alle First Class Entities die zu einer Modellinstanz gehören. Dabei wird strikt nach den Typen für die First Class Entities unterschieden. Da jedoch alle Komponenten-Typen vom *Provides* Komponenten-Typ erben, könnten beispielsweise Basic Components als *Provides* Komponenten-Typen in den Listen von `CMEnvironment` angegeben werden. Um zu verhindern, dass versehentlich Sub-Typen der eigentlich gewünschten Typen in den Listen auftreten, prüfen die OCL-Constraints den Typ der Instanz-Entitäten in der Liste mit Hilfe von `oclIsTypeOf()`.

CMEnvironment::GrantFirstClassEntityTypeCompositeComponent

CMEnvironment::GrantFirstClassEntityTypeProvidesComponentType

CMEnvironment::GrantFirstClassEntityTypeCompleteComponentType

CMEnvironment::GrantFirstClassEntityTypeImplementationComponentType

CMEnvironment::GrantFirstClassEntityTypeImplementationComponentType

CMEnvironment::GrantFirstClassEntityTypeBasicComponent

3.5.5.6. Identifier

Identifier::idHasToBeUnique
Diese Constraint garantiert, dass das `id`-Attribut einer Modellinstanz über alle Entitäten hinweg eindeutig ist.

3.5.5.7. Schnittstellen

Interface::noProtocolTypeIDUsedTwice
Wie bei SEFFs für Basic Components können auch Protokolle zur gleichen Zeit in Form verschiedener Protokoll-Typen vorliegen. Auch hier darf ein Protokoll-Typ für ein Interface nicht mehrfach verwendet werden. Daher prüft diese Constraint, dass die Typ-IDs aller Protokolle eindeutig sind.

Interface::SignaturesHaveToBeUniqueForAnInterface

Auf einer Schnittstelle dürfen Signaturen nicht mehrfach deklariert werden. Für eine Schnittstelle muss das Tupel aus Signaturname, Parametertypen (geordnet) eindeutig sein. Parameternamen und auch Exceptions werden nicht betrachtet, da bei Aufrufen von Methoden (etwa in den Programmiersprachen C# und Java) die Parameternamen und Exceptions nicht beachtet werden. Damit würden Parameternamen und Exceptions keine Eindeutigkeit erzeugen.

In der OCL-Constraint wurde das oben beschriebene Tupel nachgebildet. OCL stellt dazu über das Schlüsselwort `Tupel` explizit Tupel zur Verfügung. Zunächst wurde ein `Bag` dieser Tupeln deklariert, wobei die Parameter-Typen einer Signatur auf eine OCL-`Sequence` (geordnete Menge) abgebildet wurden. Der `Bag` wurde mit Tupeln für alle Signaturen einer Schnittstelle befüllt. Anschließend konnte die `Unique`-Operation auf dem `Bag` von Signatur-Tupeln angewendet werden, um die Eindeutigkeit der Tupel sicherzustellen.

Entgegen der OCL-Spezifikation [67], Seite 41 lässt RSA zur Definition von Tupel-Typen keine eigenen Bezeichner zu, sondern erwartet das Schlüsselwort `Tuple(..)` für die Typ-Deklaration. Die Wertzuweisung für Tupel erfolgt dann standard-konform erneut über das Schlüsselwort `Tupel(..)`.

Signature::ParameterNamesHaveToBeUniqueForASignature

Mit der vorigen Constraint wurde sichergestellt, dass Signaturen auf einer Schnittstelle eindeutig sind. Diese Constraint prüft, dass die Verwendung von Signaturen ebenfalls für die Implementierer eines Services, der über die Signatur angeboten wird, eindeutig ist. Dazu müssen die nicht über die vorige Constraint geprüften Signatur-Parameter eindeutig sein. Auch hier wird auf die `Unique`-Methode von OCL zurückgegriffen.

3.5.5.8. Delegations-Konnektoren

Delegations-Konnektoren müssen zwei OCL-Constraints genügen. Die Variante auf der angebotenen Seite ist analog zur Variante auf der benötigten Seite. Daher werden im Folgenden nur die OCL-Constraints für die angebotene Seite betrachtet.

ProvidedDelegationConnector::innerAndOuterRoleNeedToHaveDifferentContext-IDs

Diese Constraint garantiert, dass sich die Kontext-Identifier der inneren und äußeren Rolle, die über den *Provides* Delegations-Konnektor mit einander verbunden sind, unterscheiden. Wären die Kontext-Identifier identisch und damit die Kontexte identisch, entspräche dies einer infiniten Rekursion, bei der Aufrufe auf der Schnittstelle der angebotenen Rolle immer wieder auf die eigene Instanz geleitet würden.

ProvidedDelegationConnector::InnerContainingComponentEqualOuterProvidingComponent

Ein angebotener Delegations-Konnektor kann nur sinnvoll zwischen zwei angebotenen Kontext-Rollen verbinden, wenn die anbietende *Komponente* der äußeren Kontext-Rolle der *Komponente* entspricht, in der die innere Kontext-Rolle liegt. Diese Bedingung wird durch diese Constraint geprüft.

RequiredDelegationConnector::innerAndOuterRoleNeedToHaveDifferentContext-IDs
Analog zu OCL-Constraint für `ProvidedDelegationConnector`.

RequiredDelegationConnector::InnerContainingComponentEqualOuterProvidingComponent
Analog zu OCL-Constraint für `ProvidedDelegationConnector`.

3.5.5.9. Minimal-Bedingungen

ProvidesComponentType::AtLeastOneInterfaceHasToBeProvidedByAUsefullProvidesComponentType
Um die Sinnhaftigkeit von *Provides*-Komponenten-Typen zu garantieren, bestimmt diese Constraint, dass wenigstens eine Schnittstelle angeboten werden muss – also mindestens eine Rolle angeboten wird.

CompleteComponentType::AtLeastOneInterfaceHasToBeProvidedOrRequiredBy-AUsefullCompleteComponentType
Diese Constraint reglementiert minimale Anforderungen für CompleteComponentTypes sowie davon abgeleiteten Komponenten-Typen. Diese Komponenten-Typen müssen mindestens eine Rolle anbieten oder mindestens eine Rolle erfordern. Um festzustellen, ob es sich tatsächlich um die Komponenten-Typen in der Komponenten-Hierarchie handelt, für die diese Bedingung zutreffen soll, wird der Komponententyp zunächst über `oclIsTypeOf` festgestellt.

3.5.5.10. Allokation

ContextComponent::IfTwoContextComponentsAreDeployedAtDifferentCalculating-ResourcesTheAssemblyConnectorsInBetweenHaveToBeDeployedAsWell
Über diese Constraint wird garantiert, dass für zwei Kontext-Komponenten, wenn sie auf verschiedenen berechnenden Ressourcen allokiert werden, alle Assembly Konnektoren, die zwischen diesen beiden Kontext-Komponenten verlaufen, ebenfalls (auf nicht-berechnenden Ressourcen) allokiert werden.

Dazu wird für je zwei Kontext-Komponenten zunächst bestimmt, ob sie auf verschiedenen Ressourcen allokiert wurden. Ist dies der Fall, wird eine nähere Prüfung vorgenommen (`implies`). Zunächst werden jene Assembly Konnektoren bestimmt, die direkt zwischen den beiden Komponenten verlaufen. Dies sind – vereinfacht beschrieben – jene, die in folgender Menge liegen:

$$AC_{result} := Component_1 :: ProvidedRole :: AssemblyConnector$$
$$\cap\ Component_2 :: RequiredRole :: AssemblyConnector$$

wobei $Component_1$ und $Component_2$ je zwei Kontext-Komponenten unter allen Instanzen der Kontext-Komponenten sind, für die die Ressourcen, auf denen sie allokiert sind, paarweise verschieden sind. $:: ProvidesRole :: AssemblyConnector$ und $:: RequiresRole :: AssemblyConnector$ ist die Navigation zu den von den Kontext-Komponenten direkt angebotenen beziehungsweise benötigten Kontext-Rollen und von dort aus zu den assoziierten Assembly Konnektoren. Da alle Paare für $Component_1$

und *Component*$_2$ getestet werden, reicht der Test mit der angebotenen Rolle auf *Component*$_1$ und der benötigten Rolle auf *Component*$_2$ aus.

Dann ist AC_{result} eine Menge derer Assembly Konnektoren, die direkt zwischen *Component*$_1$ und *Component*$_2$ verlaufen. Für die in dieser Menge liegenden Assembly Konnektoren muss schließlich das Attribut `nonCalculatingResource` mit `notEmpty()` überprüft werden. Ist das Attribut gesetzt, wurde der entsprechende Assembly Konnektor allokiert – der Assembly Konnektor ist korrekt allokiert.

Bei der Abfrage auf Assembly Konnektoren, die über eine *Required* Kontext-Rolle erreicht werden können, musste zusätzlich eine Prüfung auf den Komponenten-Typ ergänzt werden. Nur Sub-Typen von `CompleteComponentType` verfügen über verbindliche benötigte Kontext-Rollen. Daher erfolgt ein Type-Cast auf `CompleteComponentType`, mit dem die OCL-Abfrage auf *Requires* Kontext-Rollen erfolgen kann, erst nach einer Typ-Prüfung mittels `oclIsKindOf(CompleteComponentType)`.

3.5.5.11. Komponenten-Hierarchie

Die in diesem Kapitel beschriebenen OCL-Constraints sind die komplexesten Prüfungen, die für das Komponentenmodell entwickelt wurden. Sie beschreiben eine strengere Prüfung auf den Schnittstellen von Komponenten-Typen, als das Komponentenmodell sie selbst erfordert und formalisieren damit die `conforms`-, `impl-conforms` und `realizes`-Beziehungen zwischen den Komponenten-Ebenen. In diesem Bereich tangieren die OCL-Constraints den Bereich der Validitätsprüfung des Komponentenmodells. Grundsätzlich sind Validitätsprüfungen jedoch kein Bestandteil des Komponentenmodells selbst, sondern Aufgabe externer Algorithmen. Die hier angegebenen und beschriebenen OCL-Constraints sind daher als ein *Proof-of-Concept* für die Anwendung von OCL zu sehen. Für die Weiterverwendung des UML2-Modells (inklusive der durch Transformation erzeugten Modelle), würde man eine Version ohne diese stark reglementierenden OCL-Constraints wählen.

In diesem Bereich wurden die Schwächen und Einschränkungen von OCL besonders deutlich. So ist es mit OCL nicht möglich, eine globale und / oder austauschbare Definition für das komplexe „conforms" zu hinterlegen. Dies wäre beispielsweise über Konstrukte wie OCL-Klassen oder OCL-Methoden möglich. OCL bietet jedoch nicht die Möglichkeit, Klassen mit Methoden zu definieren.

Einschränkend kam hinzu, dass RSA derzeit keine Unterstützung für das Schlüsselwort `def` bietet, das in OCL spezifiziert ist. Mit `def` sind prinzipiell globale Variablen / Operationen möglich. Daher konnten Variablen und Operationen lediglich lokal definiert werden.

CompleteComponentType::providedInterfacesHaveToConformToProvidedType

Mengenbeziehung Wie bereits angedeutet, prüft diese Constraint die Einhaltung der `conforms`-Beziehung zwischen den angebotenen Schnittstellen des *Provided Component Types* (als Menge von Schnittstellen betrachtet: I_{PT}) und den angebotenen Schnittstellen des Complete Component Types (als Menge von Schnittstellen betrachtet: I_{CT}). Dabei muss gelten $I_{CT} \supseteq I_{PT}$ – Instanzen des *Complete Component Type* müssen also mindestens die gleichen Schnittstellen anbieten, wie der *Provided Component Type*.

Die Schnittstellen-Beziehungen eines Komponenten-Typs können über die angebotenen beziehungsweise benötigten Rollen ermittelt werden.

Es muss beachtet werden, dass Schnittstellen unter einander erben können, weshalb die Prüfung neben direkten Eltern-Schnittstellen ebenfalls auch auf alle „Ahnen“-Schnittstellen (also aus der transitiven Eigenschaft der Vererbung resultierenden Eltern-..-Eltern-Schnittstellen) ausgeweitet werden muss.

Zu einer gegebenen Schnittstelle mussten dazu mit Hilfe von OCL alle „Ahnen“-Schnittstellen (im Folgenden auch *Ancestor-Interfaces*) ermittelt werden. Eine vereinfachte Prüfung mit Hilfe von `oclIsKindOf(..)` schied aus, da sich eine solche Prüfung auf die statischen Vererbungsstrukturen des Komponenten-Meta-Modells bezöge, nicht jedoch auf die dynamische Vererbungsstruktur innerhalb von Instanzen des Komponenten-Meta-Modells, die ihre Eltern-Schnittstellen über die `parentInterface`-Beziehung referenzieren.

Werden für den *Provided Component Type* alle Ahnen-Schnittstellen als Menge von Schnittstellen ermittelt, genügt die oben angeführte Prüfung auf die Obermengenbeziehung zwischen $I_{CT_{incl.\ ancestors}} \supseteq I_{PT}$ um festzustellen, ob die `conforms`-Beziehung gilt. Da der Complete Component Type ebenfalls Sub-Typen der Schnittstellen des *Provides Component Types* anbieten darf, reicht es, wenn in der Menge der Ahnen-Schnittstellen des *Complete Component Types* alle Schnittstellen des *Provides Component Types* enthalten sind.

OCL-Umsetzung Um mit Hilfe von OCL alle „Ahnen“-Schnittstellen ermitteln zu können, musste eine Rekursion in OCL realisiert werden. Dies setzt voraus, dass die Vererbungsstruktur der Schnittstellen in Instanzen des Komponentenmodells einen azyklischen Typenbaum erzeugt. Andernfalls würde eine unendliche Rekursion eintreten. Das Komponentenmodell selbst fordert ebenfalls einen azyklischen Typenbaum für die Vererbung von Schnittstellen, so dass an dieser Stelle keine Einschränkungen auf Grund von OCL gemacht werden müssen.

Da OCL keine primitiven Operatoren hat, mit denen die transitive Hülle einer Relation berechnet werden kann, im Speziellen die transitive Hülle der Ahnen-Schnittstellen zu einer gegebenen Schnittstelle, wurde auf eine rekursive Ermittlung der Ahnen-Schnittstellen zurückgegriffen. Die Möglichkeit mittels OCL Rekursionen zu beschreiben, wird in [13], S. 10ff erläutert.

Schnittstellen müssen jeweils ihre direkten Eltern-Schnittstellen mittels des `parentInterface`-Attributs deklarieren. Zusätzlich musste für Schnittstellen das `ancestorInterfaces`-Attribut eingeführt werden, das ebenfalls von Schnittstelle auf Schnittstelle verweist. Dieses Zusatzattribut wird lediglich für die Rekursion mittels OCL benötigt. Entsprechend ist das Attribut `private`. Instanzen sollten dieses Attribut nicht belegen.

Die eigentliche Rekursion wurde mittels des `let`-Schlüsselwortes realisiert. Auf der einen Seite wird die direkte Eltern-Schnittstelle in `parentInterface` gehalten, auf der anderen Seite, über ein zweites `let`, werden die bis zur aktuellen Rekursionstiefe ermittelten Ahnen-Schnittstellen mittels eines `union(..)` akkumuliert.

Die Grobstruktur der Rekursion sieht damit wie folgt aus (vgl. [13], S. 11):

```
let parentInterfaces = self.parentInterface in
  let ancestorInterfaces = self.parentInterface -> union(
    self.parentInterface.ancestorInterfaces
```

```
) in
  <some_expression_using_definition_of_ancestorInterfaces>
```

Die tatsächliche Navigation zwischen Komponenten und den letztlich angebotenen Schnittstellen erfolgt über den `iterator`-Operator. Auf den über den Iterator und die Rekursion ermittelten Mengen von Schnittstellen für den *Provided Component Type* und den *Complete Component Type* erfolgt schließlich die Prüfung der Obermengen-Beziehung mittels des `includesAll(..)`-Operators.

ImplementationComponentType::providedInterfacesHaveToConformToCompleteType

Für diese Constraint konnte eine exakte Kopie der vorangehenden Constraint verwendet werden. Da alle Attribute, unabhängig von der Typ-Ebene, gleich benannt wurden, sind keine Änderungen an der kopierten Constraint notwendig. Die Verwendung einer zentral hinterlegen Constraint für beide Ebenen schließt sich, wie bereits erwähnt, mit der Verwendung von RSA aus.

ImplementationComponentType::RequiredInterfacesHaveToConformToCompleteType

Diese Constraint funktioniert analog zu den vorangehenden Constraints, prüft jedoch entsprechend der Vorgaben des Komponentenmodells die Obermengenbeziehung zwischen den Schnittstellen-Mengen in der entgegengesetzten Richtung ab. Zwischen der Menge der benötigten Schnittstellen des Complete Component Types (I_{CT}) und der Menge der benötigten Schnittstellen des Implementation Component Types (I_{IT}) muss gelten: $I_{IT} \subseteq I_{CT_{incl.\ ancestors}}$. Auf der *Requires*-Seite muss also für den *Vater*-Komponenten-Typ die Menge der Ahnen-Schnittstellen bestimmt werden.

BasicComponent::ProvideSameInterfaces

Über diese Constraint wird sichergestellt, dass die Basic Component des Kontexts die gleichen Schnittstellen über `ProvidedRoles` anbietet, wie der über die mit `realizes` stereotypisierte Relation erreichbare `implementationComponentType`. Die Menge der angebotenen Rollen wird dabei nur überprüft, wenn die Assoziation zum Implementation Component Type nicht leer ist.

BasicComponent::RequireSameInterfaces

Analog zur *Provides*-Seite prüft diese Constraint auf der *Requires*-Seite die Einhaltung der `realizes`-Relation ab.

CompositeComponent::ProvideSameInterfaces

Diese Constraint ist in Analogie zur Prüfung der angebotenen Schnittstellen für Basic Components (siehe oben) aufgebaut.

CompositeComponent::RequireSameInterfaces

Diese Constraint ist in Analogie zur Prüfung der benötigten Schnittstellen für Basic Components (siehe oben) aufgebaut.

3.5.5.12. Erfahrungen mit der Verwendung von OCL

Literatur Im vorangehenden Kapitel wurden Rekursionen mittels des `let`-Schlüsselwortes definiert. Wie in [13], S. 11 jedoch festgestellt wird,

> „However the construct's semantics within OCL is not entirely clear [.. [24]]. Whether arbitrary recursively defined expressions are allowed is uncertain. Thus, using let to define transitive closure is not advised.“,

ist durch die OCL-Spezifikation [67] nicht festgelegt, wie die genaue Semantik von `let` im Bezug auf Rekursionen definiert ist. Auch ganz allgemein offenbart die OCL-Spezifikation einige Lücken im Bezug auf präzise Semantik. So schlagen Brucker und Wolff [16, 17] eine formale Semantik für OCL („HOL-OCL“) vor, die sie aus der Einbettung von OCL in *High-Order Logic* gewonnen haben. Die Universität Kent, die eine (partielle) Implementierung [3] für den OCL 2.0 Standard bietet, stellt insgesamt 17 Einschränkungen der OCL-Spezifikation fest ([2] S. 37ff). So führen Akehurst und Patrascoiu von der Universität Kent ([3], S. 16) an:

> „Our experience has illustrated many areas in which the standard requires improvement and we have provided ideas to address some of these improvements“.

Sie beziehen sich damit vornehmlich auf eine fehlende Referenz-Implementierung von OCL, die die Spezifikation aufwerten und präzisieren würde. Daneben beziehen sich die gleichen Autoren in [2], S. 37ff beispielsweise auf eine fehlende Semantik und Syntax für die Basis-Datentypen Integer, Real und String oder die unscharfe Abgrenzung der Operatoren `oclIsUndefined()` und `isEmpty()`.

Insgesamt wird deutlich, dass OCL auch in der Version 2.0 noch Lücken bei der Präzision der Semantik aufweist. Konkrete Konzepte zur Präzisierung der OCL-Spezifikation werden vor allem in [2] aufgezeigt.

Fazit Die eigene praktische Erfahrung mit OCL zeigt: OCL ist gut geeignet, um einfache Bedingungen formal niederzuschreiben. Unter RSA büßt OCL jedoch Übersichtlichkeit bei umfangreicheren Constraints ein, da das `def`-Schlüsselwort nicht unterstützt wird und damit keine kontext-globalen Definitionen von Operationen und Variablen möglich sind. Zudem lässt die Benutzeroberfläche von RSA lediglich vier Zeilen sichtbaren OCL-Codes zum Zwecke der Bearbeitung zu. Dies schränkt den Komfort der Bearbeitung umfangreicher Constraints erheblich ein.

Auch in der Spezifikation von OCL ist der Nutzen von `def` eingeschränkt. Definitionen mittels `def` sind immer nur in einem Kontext gültig, nie global auf einem kompletten UML2-Modell. Daher ist eine Wiederverwendungen von Operationen und Variablen, unter der Annahme, dass dies vom Modellierungs-Werkzeug unterstützt würde, in OCL nur innerhalb *eines* Kontexts möglich. Wünschenswert wäre die Möglichkeit, dass `def`s global gültig wären. Im Komponentenmodell wäre die globale Definition von Operationen beispielsweise für die Definition der *conforms*-Beziehung angenehm gewesen.

Insbesondere Rekursionen in OCL sind wenig intuitiv über das `let`-Konstrukt definierbar, da die Rekursion über die Ersetzung von Variablen geschieht. Nehmen zudem auf dem UML2-Modell navigierende Operationen – etwa mittels des OCL-Iterators –

zu, geraten OCL-Ausdrücke insgesamt unübersichtlich und schwer verständlich, weil OCL-Iteratoren dann in mehreren verschachtelten Schleifen verwendet werden.

Der Iterator unter OCL verlangt zwingend, dass der Akkumulator-Teil des Iterators definiert und verwendet wird. Eine reine Verwendung als Iterator ohne Nutzung des Akkumulators, der auch als Rückgabemenge dient, ist nicht vorgesehen. In vielen Szenarien wäre ein Iterator wünschenswert, der lediglich iteriert ohne zusätzlich zwingend Mengenoperationen durchzuführen. Der derzeitige OCL-Iterator stellt ein Mischkonzept aus „klassischem" Iterator und einer Manipulation auf Mengen dar.

So wurde der Iterator bei der Navigation über Mengen von Mengen genutzt. Auch wenn nur in der innersten Menge Ergebnis-Elemente gefunden wurden, die über den Akkumulator zurückgeliefert werden sollten, mussten die äußeren Akkumulatoren zwingend verwendet werden – ohne das etwa Mengenoperationen durchgeführt wurden.

3.5.6. Transformation von OCL-Constraints

Das UML2-Modell des Komponenten-Meta-Modells samt definierter OCL-Constraints wurde bis jetzt dargestellt. Damit wurde der erste Schritt des Prozesses aus Abbildung 3.2 („Rational Software Architect") beschrieben. Im weiteren Verlauf wird insbesondere auf die Transformationsprozesse des UML2-Modells sowie nachfolgender Repräsentationen eingegangen.

Die Transformation von OCL-Constraints mittels EMF ist eingeschränkt, wie auch in Kapitel 4.2.1.1 beschrieben wird. Um OCL-Constraints die in einem UML2-Modell definiert wurden, unter EMF weiter zu verwenden, müssen diese derzeit manuell in die von EMF erzeugten Modelle übernommen werden. In [29], S. 4 wird diese Erfahrung mit OCL und EMF gestützt:

> „Abhilfe bietet hierbei neuerdings die OCL-Programm-Bibliothek [..] der Universität Kent, mit der sich OCL-Bedingungen direkt in das EMF-Metamodell integrieren lassen. Bislang muss diese allerdings noch per Hand geschehen, da der EMF-Generator OCL noch nicht unterstützt."

Da keine manuelle Übersetzung der OCL-Constraints durchgeführt wurde, konnten die im UML2-Modell spezifizierten OCL-Constraints nicht validiert werden. RSA bot keine Möglichkeit zur Modellierung von Modell-Instanzen und die von EMF generierten Modelle boten, wie oben beschrieben, ebenfalls keine OCL-Unterstützung.

Mit dem Technologieprojekt von EMF „EMFT" [28] soll eine Unterstützung von OCL unter EMF eingeführt werden. Damit besteht zukünftig die Möglichkeit Quellcode zur Überprüfung von Modell-Instanzen generieren zu können. Für die Diplomarbeit sind diese geplanten Entwicklungen jedoch noch nicht nutzbar.

3.5.7. Einschränkungen durch die Verwendung von RSA

In Kapitel 3.5.1 wurde bereits auf den Export des UML2-Modells zu ECORE und dem anschließenden Import in EMF eingegangen. Insgesamt sind für den geplanten Prozess folgende Einschränkungen durch die Verwendung von RSA festzustellen:

- Für das UML2-Modell sollten an mehreren Stellen Stereotypen für Assoziationen verwendet werden (bspw. *conforms*, *extends*, *realizes*). Stereotypen werden

in RSA über externe UML Profile deklariert. Nach dem Import dieser UML Profile, die Stereotypen definieren, in ein UML-Modell unter RSA, können in RSA selbst die Stereotypen verwendet werden. Der Export von RSA in das UML2-Serialisierungsformat schlägt mit der Verwendung von UML Profilen jedoch fehl. Bei komplexeren Modellen erfolgt durch den Export keine Ausgabe mehr in das Dateisystem.

Als *Workaround* wurden die Stereotypen der Assoziationen als *Label* hinterlegt. Unter RSA und in UML2-Modellen allgemein müssen *Label* von Assoziation jedoch innerhalb eines Diagramms eindeutig sein. Daher wurden die Label, die Stereotypen repräsentieren, nach dem folgenden Muster aufgebaut: ≪`[Stereotype]-[UsageNumber]`≫, wobei „UsageNumber“ eine Zahl ist, die sich je Verwendung des gleichen Stereotyps unterscheidet.

Anmerkung: Der Export *kleinerer* Modelle, die Stereotypen über UML Profile verwenden, gelingt, sofern UML2 als Format gewählt wird. Beim Export zu ECORE werden die Stereotypen überhaupt nicht beachtet.

- Werden in OCL-Constraints verschachtelte `lets` verwendet, so müssen in der äußeren Verschachtelung definierte Variablen ebenfalls im *Body* der inneren `lets` verwendet werden, auch wenn die äußere definierte Variable unter Umständen lediglich zur Initialisierung der Werte eines inneren `lets` verwendet werden müsste. Wird die in der äußeren Verschachtelung definierte Variable im Inneren nicht verwendet, liefert die eingebaute OCL-Validierung von RSA eine Meldung über fehlerhaftes OCL.

- RSA lässt für OCL-Constraints keine manuelle Definition des Kontexts mit Hilfe des Schlüsselwortes `context` zu. RSA nimmt als Kontext immer implizit die UML2-Klasse an, der die Constraint zugeordnet ist.

- RSA bietet keine Unterstützung für das `def`-Schlüsselwort (siehe oben).

- Ab der ersten Mengen-Operation auf einem Typ der OCL funktioniert die *Auto-Completion* nicht mehr für weitere angehängte Ausdrücke auf den Ergebnismengen von Mengen-Operationen.

- Beim Export von RSA zum UML2-Serialisierungsformat (`.uml2`), verletzt RSA den XMI-Standard [66], um andere Modelle zu referenzieren (siehe [42]). Referenzen auf andere (externe) Modelle, die importiert werden sollen, werden mit der *RSA-internen* Adressierung angegeben. Referenziert etwa „Model2“ die Klasse „Class1“ aus „Model1“, erfolgt der Verweis mit:

  ```
  href=''Model1.emx#_DCnYQJ1-Edqn2aG2G5-fdg?Model1/Class1''
  ```

  Leicht zu erkennen ist die Endung `.emx`, der Dateiendung für das Serialisierungsformat von RSA.

- Beim Export von UML2-Modellen, die OCL-Ausdrücke verwenden, aus RSA heraus bestehen signifikante Unterschiede, ob der Export zu ECORE oder UML2 als Serialisierungsformat geschieht. Während der Export zu UML2-Modellen vollständig erfolgt, ist der ECORE-Export von OCL-Ausdrücken nicht vollständig.

> In UML2 werden die Bezeichnung der Sprache für Regel-Ausdrücke („OCL") ebenso hinterlegt, wie der eigentliche `Body`-Text des OCL-Ausdrucks, ein Verweis auf die Entität des UML2-Modells, auf den sich die Regel bezieht, und der Name der OCL-Constraint.
>
> Der Export zu ECORE führt zu einer Verstümmelung von Constraints. Dem ECORE-Standard (analog zu [65]) entsprechend werden Constraints in `eAnnotations` abgelegt und dann über das `details`-Element mit dem `key` als Constraint deklariert. Zusätzlich wird der Name der Constraint als `value` abgelegt. Die `eAnotation` selbst wird in der XML-Struktur als Kind-Knoten der Klasse abgelegt, für die die Constraint gilt. Insgesamt lässt sich damit rekonstruieren, für welche UML2-Klassen OCL-Constraints mit welchem Namen gelten. Der *Body*-Text des OCL-Ausdrucks geht jedoch über den Export (so wie er von RSA angeboten wird) verloren. ECORE böte gleichwohl die Möglichkeit den Text des OCL-*Bodies* zu definieren.

Insbesondere im Bezug auf OCL erscheint RSA derzeit noch nicht vollständig ausgereift. So ist die Editierunterstützung weder besonders komfortabel, wenn es darum geht größere OCL-Constraints zu bearbeiten, noch funktioniert der Export in allen Fällen vollständig. Beide möglichen Varianten des Exports des UML2-Modells aus RSA heraus, ECORE und UML2 (andere Export-Formate sind mit EMF nicht verwendbar), sind nicht perfekt. Notwendig wäre die Möglichkeit, sich gegenseitig referenzierende Modelle exportieren zu können. Wie zu Beginn des Kapitels zur Modellierung geschildert wurde, teilt sich PalladioCM in weitere Sub-Modelle auf. Zusätzlich wäre notwendig, dass OCL-Constraints vollständig exportiert würden.

Der Export zu ECORE bietet sich gegenseitig referenzierende Modelle, aber keine vollständige Unterstützung für OCL-Constraints. Der Export zu UML2 bietet eine vollständige Unterstützung für OCL-Constraints, ohne jedoch mit verteilten Modellen zu funktionieren.

Trotz der genannten Einschränkungen gab es aus Lizenzgründen keine Alternative zur Verwendung RSA. Für Borland Together liegt keine gültige Lizenz zur Verwendung in der Diplomarbeit vor, freie UML-Tools reichen (soweit bekannt) nicht an den Funktionsumfang von RSA heran, würden also zusätzliche Einschränkungen bedeuten.

3.5.8. Weitere Einschränkungen bei der UML2-Modellierung

Das interne Format von EMF, ECORE, kennt zur Auszeichnung von Identifiern das *Tag* `isID`. Dieses kennzeichnet Attribute als eindeutige Identifier. In UML2 existiert ein solches *Tag* jedoch nicht. Daher kann die Klasse `Identifier`, die letztlich genau diese Funktion inne hat, nicht als solche gekennzeichnet werden. Unter anderem aus diesem Grunde machte es zusätzlich Sinn, Identifier als eigenes UML2-Modell von `PalladioCM` abzutrennen. Auf diese Weise kann die einmal exportierte ECORE-Serialisierung von *Identifier* gezielt manipuliert werden. Da *Identifier* eine deutlich kleinere Änderungshäufigkeit zu erwarten hat, als das gesamte, deutlich komplexere, `PalladioCM`, fällt der Änderungsaufwand insgesamt kleiner aus, wenn das `isID`-Tag nachträglich manuell gesetzt wird.

Attribute von Klassen dürfen in UML2-Modellen nicht mit `context` benannt werden, da dies ansonsten zu einem Benennungskonflikt mit dem `context` aus OCL führt.

OCL erlaubt grundsätzlich keine Mehrdeutigkeiten und nimmt (in der RSA-Implementierung) bei `context` immer das Schlüsselwort an. Im Klassen-Diagramm „Context“ wurden daher Attribute, die auf den Komponentenmodell-Kontext verweisen, in `parentContext` umbenannt.

4. Eclipse Modeling Framework

Im vorangehenden Kapitel wurde die Modellierung des Komponenten-Meta-Modells in UML2 sowie der Export dieses Modells in Zwischenformate diskutiert. In diesem Kapitel geht es nun darum, wie das Komponenten-Meta-Modell in EMF importiert wird und welche Transformationen dabei durchgeführt werden müssen. Zudem wird behandelt, welche weiteren Transformationen mit Hilfe von EMF zur Erzeugung von Java-Quellcode möglich sind. Schließlich soll es darum gehen, welche Erweiterungen am ECORE-Modell in EMF durchgeführt werden mussten, damit dieses Modell den Ansprüchen des Komponenten-Meta-Modells und den Konzepten dahinter genügt.

4.1. Import von Serialisierungs-Artefakten

Für den Import von Serialisierungs-Artefakten in EMF, die mit Hilfe von RSA erzeugt wurden, boten sich grundsätzlich UML2 oder ECORE als Formate an. Im vorigen Kapitel wurden bereits Einschränkungen für den Export genannt. Da es für die Diplomarbeit grundsätzlich wichtiger war, ein vollständiges UML2-Modell weiterzuverarbeiten, in dem möglichst viele Konzepte enthalten sind, schied UML2 als Zwischenformat aus. Das UML2-Modell war so angelegt, dass es auf verschiedene Sub-Modelle verteilt war. Solche verteilten Modelle wurden von RSA nicht korrekt exportiert und konnten daher in EMF nicht importiert werden.

Für zukünftige Versionen von RSA und EMF können jedoch andere Einschränkungen gelten. Daher erfolgt eine Abwägung der Vor- und Nachteile der beiden Serialisierungsformate bezüglich des Imports in EMF.

4.1.1. UML2-Format

- Erfolgt der Import aus einer UML2-Datei, werden OCL-Constraints vollständig als `EAnnotation` in das ECORE-Modell von EMF importiert.

- Beim Import erfolgt eine einfache Validitätsprüfung des importierten Modells. Geprüft wird auf doppelt vererbte Attribute, doppelte Vererbung, etc.

- Werden Entitäten importiert, die einen nicht validen Java-Namen haben (etwa mit einem „-“ im Namen), erfolgt eine Umbenennung.

- Der Import verlinkter, sich referenzierender, Modelle (die zuvor aus RSA fehlerhaft exportiert wurden) ist nicht möglich und bricht mit einer Fehlermeldung ab.

4.1.2. ECORE-Format

- Werden untereinander verlinkte und sich referenzierende Modelle importiert (wie beispielsweise „PalladioCM“ und „FSM“), so werden gegenseitige Abhängigkeiten erkannt. Der Import erfolgt nur, wenn alle benötigen Modelle zugleich importiert werden.

- Eine Umbenennung nicht Java-konformer Namen von Entitäten erfolgt nicht.

- Eine sonstige Validitätsprüfung erfolgt nicht.

- OCL-Constraints werde nur unvollständig importiert.

4.2. Transformationen beim Import

Im vorangehenden Kapitel wurden bereits die zur Auswahl stehenden Alternativformate für den Import vorgestellt. Die Modellierung, die in einem UML2-Modell mündete, resultierte in mehreren sich referenzierenden Modellen. Da diese von RSA nur über das ECORE-Format unterstützt werden, beziehen sich die im Folgenden geschilderten Import- und Transformationsvorgänge auf dieses Format.

Der Import-Vorgang ähnelt dem in [31] beschriebenen Vorgehen. Für den Import in EMF wurde zunächst ein „EMF Project“ unter Eclipse angelegt. Dabei konnte zwischen den zu importierenden Sub-Modellen des Komponentenmodells (Domänenmodell) gewählt werden. Die ECORE-Dateien werden ohne Transformationen importiert, da es sich bei ECORE zugleich um das interne Datenformat von EMF handelt (im Gegensatz zum UML2-Format). EMF integriert automatisch alle importierten ECORE-Dateien (-Modelle) in das erzeugte Projekt.

EMF erzeugt aus einem in ECORE vorliegenden Domänenmodell Java-Quellcode, der das ECORE-Modell repräsentiert. Der Kern der Generierungsfunktion von EMF spiegelt sich in einer zentralen `.genmodel`-Datei wider. In dieser können Anweisungen für die Transformation hinterlegt werden. Die genauen Regeln, denen diese Transformation folgt, können damit zentral gesteuert werden.

Zu den steuerbaren Eigenschaften zählen unter anderem:

- Festlegung von Namensräumen

- Die Editierbarkeit von Attributen

- Die Einrichtung von Notifikationsmechanismen (automatisiertes *Event-Handling*)

- Erzeugung von *Factories*

- Erzeugung von Unit-Testfällen

Die Trennung von Modell und Transformation wird dabei nicht verletzt. Transformationsanweisungen (`.genmodel`) werden getrennt vom Modell (`.ecore`) gespeichert.

4.2.1. Erzeugung von Java-Code

EMF bietet eine Unterteilung der Erzeugung von Java-Code zum Domänenmodell in vier Bereiche: Modell, Edit, Editor und Test.

Nilesh Kawane geht in seiner *Master Thesis* (vgl. [41], S. 31ff) auf die Erzeugung von Programm-Code aus UML Klassendiagrammen ein, die bei der Erzeugung des oben genannten Bereichs „Modell" Anwendung findet.

4.2.1.1. Modell

Der erste Teil betrifft die Erzeugung von Code für Modelle. Zu jedem ECORE-Modell, das in EMF importiert wurde, werden drei Pakete erzeugt. Das erste Paket enthält Schnittstellen zu allen Entitäten, die in dem ECORE-Modell definiert wurden. An dieser Stelle wird klar, weshalb eine Modellierung des UML2-Modells unter RSA nicht mit Hilfe von Schnittstellen erfolgen muss. Da aus jeder Klasse (Entität) im UML2-Modell eine ECORE-Klasse wird und dazu schließlich eine Schnittstelle erzeugt wird, ist es nicht notwendig bereits im UML2-Modell über Schnittstellen zu modellieren. Jede Klasse des UML2-Modell entspricht in Folge dieses Transformationsschrittes einer Meta-Klasse des ECORE-Modells.

Zusätzlich legt EMF ein *Interface* an, das das *Package* der *Interfaces* mit den Metadaten zu Name und Namensraum beschreibt sowie Zugriff auf die Meta-Objekte aller Klassen, Methoden jeder Klasse, Enumeratoren und aller Datentypen bietet.

Instanzen werden in EMF auf Wunsch durch Factories erzeugt. Je Modell wird ein *Factory-Interface* angelegt, das die Erzeugung von Modell-Entitäten ermöglicht.

Das zweite Paket `.impl` bietet eine Implementierung der *Interfaces*, die gerade beschrieben wurden. Darin wird unter anderem ein Notifikationsmechanismus generiert, der Änderungen an jeglichen Attributen registriert und über eine `NotificationChain` publiziert. Zusätzlich werden (beim Import des UML2-Serialisierungsformats) Code-Rümpfe zur Constraints-Prüfung angelegt. Diese zunächst deaktivierten Prüfmethoden, die je Constraint des importierten Modells erzeugt werden, ermöglichen es, dass einer `DiagnosticChain` Informationen über fehlgeschlagene Prüfungen mitgeteilt werden und liefern zudem einen boolschen Wert über gültige und ungültige Constraints zurück. Quellcode, der beispielsweise die mittels OCL spezifizierten Constraints überprüft, wird nicht generiert.

Das dritte Paket `.util` stellt, dem Namen entsprechend, Hilfsklassen zur Verfügung. `Switch` stellt eine dem Prototyp-Muster (vgl. [34], S. 144ff) ähnliche Funktionalität bereit, die von `AdapterFactory` genutzt wird. Über letztere lassen sich vereinfacht Adapter (vgl. [34], S. 171ff) zu allen Entitäten eines Modells erzeugen.

Allgemein dient dieser erste Teil der Erzeugung von Code auch dazu, die für ein Eclipse-Plugin notwendigen `properties`-Dateien automatisch durch EMF anlegen zu lassen. Dadurch brauchen diese Konfigurationsdateien für Plugins nicht notwendigerweise von Hand angepasst zu werden, um dennoch ein lauffähiges Eclipse-Plugin erzeugen zu können.

4.2.1.2. Edit

Der zweite Teil der Code-Erzeugung „Edit" legt ein eigenständiges weiteres Sub-Projekt an. Innerhalb dieses Sub-Projekts werden je Modell Pakete angelegt, die `[Modelname].`

provider benannt sind. Hierin werden Implementierungen für so genannte ItemProvider generiert, die Adapter für die Modellklassen darstellen und zum Editieren und zur Anzeige verwendet werden. Die Erzeugung von Instanzen dieser Adapter übernimmt wiederum eine *Factory*.

4.2.1.3. Editor

Den dritten Teil generierbaren Codes stellt der „Editor“ dar. Dieser Editor beinhaltet unter anderem eine GUI zur Darstellung und Bearbeitung von Modell-Instanzen in einer Baumansicht. Damit lassen sich Instanzen der ECORE-Modelle in einer einfachen Form manipulieren.

Je Modell werden drei Java-Pakete angelegt, die sich wie folgt aufteilen:

- `ActionBarContributor` stellt Einträge für die grafischen (Kontext-)Menüs des Editors bereit. Hierzu zählen Einträge zum Generieren neuer Entitäten.
- `Editor` stellt den eigentlichen Editor dar.
- `ModelWizard` stellt einen einfachen *Wizard* zum Erstellen neuer Modelle bereit. In diesem *Wizard* kann eine initiale Entität gewählt werden, die als Ausgangspunkt für die Bearbeitung mit dem Editor dient.

In Abbildung 4.1 wird der über EMF generierte Editor für eine beispielhafte Modell-Instanz gezeigt. Im oberen Bereich ist die Baumansicht der Modell-Instanz zu sehen. Eine Schnittstelle mit einer Annotation und Signaturen wird über eine Provided Rolle von einer Provides Component angeboten. Im unteren Bereich des Bildschirmfotos werden die Eigenschaften der selektierten Schnittstelle angezeigt. Die angegebene `Id` wurde durch den Editor generiert. Zusätzlich ist erkennbar, dass die Schnittstelle lediglich durch eine Rolle angeboten wird.

Über diesen Editor sind die ECORE-Modelle, die insgesamt das Komponentenmodell bilden, vollständig bearbeitbar. Auch dynamisch referenzierbare Aspekte wie FSMs oder Petri-Netze lassen sich mit Hilfe des Editors in Form von Protokollen oder SEFFs verwenden. Zusätzlich funktioniert eine vereinfachte Form der Konsistenzprüfung. Diese verwendet keine OCL-Constraint, prüft jedoch über die ECORE-Modelle selbst definierte Bedingungen, wie die zwingend notwendige Existenz einer Instanz von `CMEnvironment` (die im Beispiel nicht enthalten ist).

4.2.1.4. Test

Der vierte Teil generiert schließlich Codestümpfe für Unit-Tests zu den Entitäten der Modelle. Zusätzlich wird ein Beispiel-Test je Modell generiert, der Unit-Tests für den von EMF erzeugten Java-Code illustriert.

4.2.2. Berücksichtigung von Constraints unter EMF

Für den oben geschilderten Fall des Imports von ECORE-Modellen in EMF findet keine Berücksichtigung von (OCL-)Constraints statt. Anders verhält sich EMF für den Import von UML2-Modellen. Aus Gründen der Vollständigkeit sollen daher die Möglichkeiten zur automatischen Generierung von Constraint-Code dargestellt werden.

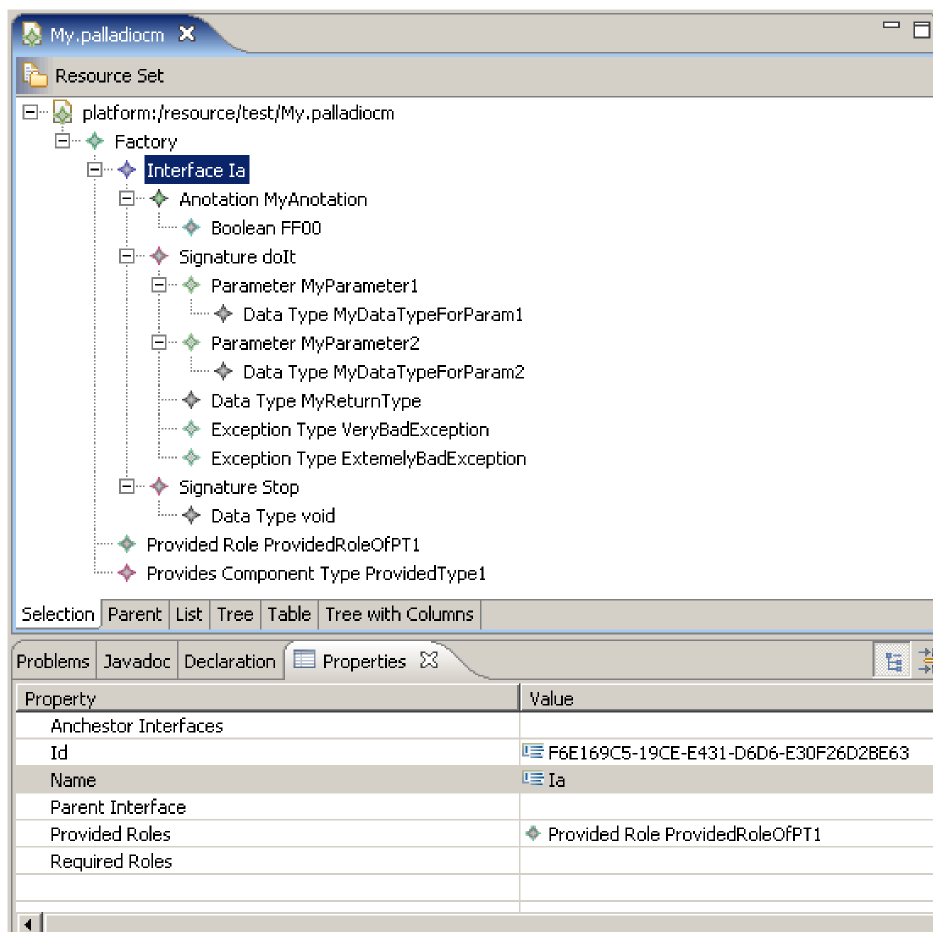


Abbildung 4.1.: Generierter EMF Editor mit beispielhafter Modell-Instanz

EMF selbst bietet keinerlei Unterstützung für *OCL*-Constraints. Im Gegensatz dazu hat das Technologieprojekt zu EMF „EMFT“ [28] dieses Manko erkannt und arbeitet für zukünftige EMF-Versionen an einer integrierten Unterstützung von OCL. Gleichwohl lassen sich Constraints (im Allgemeinen) in EMF formulieren – EMF unterstützt Implementierer durch die Erstellung von Code-Gerüsten.

Als Code-Gerüst wird zu jeder Entität einer ECORE-Klasse und zu jeder Constraint eine Methode mit boolschem Rückgabewert angelegt. Die Methode trägt den Namen der (OCL-)Constraint, wie sie im ECORE-Modell angegeben wurde. Der Methodenrumpf ist zunächst unmittelbar nach dem Generieren über ein `if(false)` deaktiviert, kann jedoch einfach aktiviert und mit prüfendem Java-Code versehen werden. EMF bietet überdies eine Art „Diagnosekette“, einen Mechanismus, der es erlaubt eine Menge von Fehlermeldungen strukturiert abzulegen. Entsprechender Java-Code wird bereits durch EMF generiert.

Um dennoch mit der aktuellen Version von EMF OCL-Constraints prüfen zu können, ist ein Rückgriff auf Dritt-Bibliotheken notwendig. Als Standard-Bibliothek hat sich dabei die „Kent OCL Library“ [74] der Universität Kent entwickelt. Ein Tutorial zur Verwendung von OCL mit EMF ist bei IBM zu finden [76].

Für den in der Diplomarbeit angestrebten Prozess war diese Form der OCL-Unterstützung zunächst nicht nutzbar. OCL hätte von Hand in den von EMF erzeugten Code eingepflegt werden müssen. Mit jeder Änderung des UML2-Modells wäre zu befürchten, dass die OCL-Unterstützung neu hätte implementiert werden müssen. Damit würde der Ansatz zur automatischen Modelltransformation unterlaufen werden. Der Gesamtaufwand für den Durchlauf eines Generierungsprozesses stiege an.

4.2.3. Merge bestehender Modelle unter EMF

Im vorangegangenen Kapitel wurden die Fähigkeiten von EMF Modelle zu *mergen* bereits angedeutet. EMF – genauer die von EMF verwendete *JET-Template-Engine* – bietet lediglich einen rudimentär ausgeprägten *merge*-Mechanismus. JET unterscheidet Quellcode in generierte und nicht generierte Abschnitte. Dies wird mit den in Java üblichen Tags `@generated` beziehungsweise `@generated not` spezifiziert. Diese Tags stehen im JavaDoc-Kopf von Klassen, Methoden, Enumeratoren und Klassenvariablen. Entsprechend der Art und Weise wie eine Markierung zur Unterscheidung generierten Codes von nicht generiertem Code erfolgen kann, ist die Granularität des *merge*-Mechanismus’ ausgeprägt. EMF unterscheidet lediglich, ob Code komplett neu generiert werden soll oder ob bestehender Code erhalten bleiben soll. So unterscheidet das `@generated`-Tag auf Klassen, ob gesamte Klassen stets neu generiert werden sollen. Wird das Tag bei Methoden verwendet, entscheidet dies über die Neugenerierung der kompletten Methode; usw.

In JavaDoc selbst erlaubt EMF eine feingranularere Deklaration jener Bereiche, die von einer Neu-Generierung nicht betroffen sein sollen. Mittels `<!-- begin-user-doc -->` und `<!-- end-user-doc -->` können Bereiche der Dokumentation von der Neu-Generierung ausgenommen werden.

Sollen zwei Modell-Generationen zusammengeführt werden, von denen die frühere Generation mehr Methoden auf einer Klasse definiert als die spätere Generation, sollte erwartet werden, dass EMF nicht mehr benötigte Methoden erkennen und entfernen kann. Dies ist nicht der Fall. EMF operiert ausschließlich auf allen Modell-Elementen

der jüngeren Modell-Generation und führt nur dort Anpassungen über Neu-Generierungen durch. Code-Bereiche, die aus früheren Modell-Generationen stammen, werden komplett ignoriert – so auch nicht mehr benötigte Methoden oder Klassen.

Für den in der Diplomarbeit geplanten Prozess bedeutet dies, dass die Verwendung des Merge-Mechanismus von EMF nur eingeschränkt erfolgen sollte. Die händische Implementierung von Quellcode macht nur für jene Teile Sinn, die als wenig variabel im Komponentenmodell gelten. Steht zu befürchten, dass eine komplette Meta-Klasse des UML2-Modells häufig geändert wird und dadurch eventuell nicht mehr benötigte Methoden unerwünscht im Quellcode erhalten blieben, würde dies eine manuelle Nachpflege der entsprechenden Java-Klasse bedeuten. Daher folgt, dass variable Aspekte des Komponentenmodells von weniger variablen Aspekten getrennt werden sollten. Dies wurde bei der Modellierung des Komponentenmodells berücksichtigt, wie beispielsweise Kapitel 4.3 zeigt. Über die Schaffung getrennter Modelle lässt sich der Generierungsprozess von EMF gezielt nur auf einzelne Modelle anwenden. Dadurch können händische Veränderungen erhalten werden, während andere Bereiche des Komponentenmodells komplett neu generiert werden.

Für die Verwendung des Komponentenmodells zusammen mit EMF in der bislang dargestellten Form ist von der händischen Implementierung lediglich das Identifier-Konzept betroffen. Die Umsetzung beschreibt das folgende Kapitel.

4.3. Verwendung von Identifiern

Für die Modellierung von Identifiern (und der Weiterverwendung unter EMF) wurden im Rahmen der Diplomarbeit zwei Alternativen evaluiert. Die erste – letztlich verwendete – Alternative wurde bereits in Kapitel 3.5.4.4 vorgestellt. Die zweite Alternative ermöglicht einen einfachen Austausch der Realisierung von Identifiern durch dynamisches Referenzieren von beispielsweise GUIDs. Beiden Alternativen gemein ist, dass Identifier als eigenständiges, vom Komponentenmodell-Kern unabhängiges, Modell existieren.

Identifier wurden im Komponentenmodell eingeführt, um eine von Modell-Instanzen unabhängige eindeutige Identifikation von Modell-Entitäten-Instanzen zu ermöglichen. Damit EMF zur Serialisierung ebenfalls die vom Komponentenmodell vorgesehenden Identifier benutzt, muss dies im ECORE-Modell explizit angegeben werden. ECORE lässt als Identifier lediglich den einfachen Datentypen `EString` zu, der als Attribut einer Klasse zu definieren ist. Damit sind komplexe Datentypen (etwa eigenständige Klassen) oder andere Basis-Datentypen (`EInt`, `EDouble` usw.) als Identifier ausgeschlossen.

Ein früherer Modellierungsversuch, bei dem Identifier als eigenständige Entität existierte, die von Entitäten lediglich über eine Komposition erfasst wurde, musste aus diesem Grund aufgegeben werden. Um Identifier in eine andere Klasse auslagern zu können, ist also zwingend eine „Ererbung“ des `id`-Attributs (mit samt einer möglichen Erzeugungsstrategie) notwendig, damit diese Identifier ebenfalls in EMF genutzt werden können. Das in ECORE-Modellen zu setzende Attribut, damit ein Attribut als Identifier verwendet wird, lautet: `ID`.

4.3.1. Alternative Modellierung von Identifiern

Modellierung Dieses Kapitel beschreibt eine alternative Möglichkeit zur Modellierung von Identifiern. Im Gegensatz zu anderen Modellierungsansätzen ermöglicht dieser über die Verwendung zusätzlicher Sub-Modelle den einfachen Austausch für die Erzeugungsroutinen von Identifiern. Die Implementierung der Identifier-Erzeugung ist bei diesem Ansatz nicht fix, sondern wird über die dynamische Referenzierung eines Modells zur Erzeugung von Identifiern gesteuert.

Auch bei diesem Ansatz erbt `Entity` von `Identifier` (Abbildung A.24), der das `id`-Attribut deklariert. `Identifier` jedoch aggregiert einen `IdGenerator` (Abbildung A.25) – einer Klasse, die über die Methode `generateId()` einen neuen Identifier generiert. Dieser Wert kann von `Identifier` zur Belegung des `id`-Attributes verwendet werden. Neben `Identifier` ist auch `IdGenerator` abstrakt. Konkrete Realisierungen von `IdGenerator` zur Erzeugung von ids, wie etwa GUID (siehe Abbildung A.26), brauchen lediglich von `IdGenerator` erben, um selbst die Erzeugung von ids anbeiten zu können.

Instanzen von `IdGenerator`-Implementierungen werden bei diesem Ansatz zentral von der *Factory* des Komponentenmodells erzeugt (`Factory` komponiert dazu `IdGenerator`). Diese lässt genau eine Instanz von `IdGenerator` zu. Durch die Limitierung des Erzeugungsvorgangs auf die *Factory* wurde ein Singleton (vgl. [34], S. 157ff) geschaffen. Eine zentrale gemeinsame Instanz einer Realisierung des `IdGenerator`s sorgt zudem dafür, dass ids stets auf die gleiche Weise (und insbesondere innerhalb einer Modell-Instanz konsistent) erzeugt werden.

Der mit dieser Alternative generierte Editor ist als Bildschirmfoto in Abbildung A.27 und A.28 zu sehen. Die Baumansicht unterscheidet sich dabei nicht vom Editor, wie er in Abbildung 4.1 dargestellt wurde. Im Properties-View ist zu erkennen, dass jede Entität bei diesem Ansatz den `IdGenerator` angeben muss. Bei der praktischen Verwendung dieses Modell-Editors bedeutet dies jedoch, dass der zu verwendende `IdGenerator` für jede Entität von Hand spezifiziert werden muss. Ist der `IdGenerator` nicht gesetzt, wird keine ID erzeugt.

Um in diesem Bereich eine Komfortgewinn erzielen zu können, müsste der Instanzierungsprozess aller Entitäten, der von EMF generiert wird, manuell angepasst werden: Für jede Entität müsste bei oder nach der Instanzierung der aktuell gültige zentrale `IdGenerator` gesetzt werden. Eine solche manuelle Anpassung des generierten Java-Codes erscheint nicht sinnvoll, da dieser Code bei jeder Änderung des UML2-Modells des Komponentenmodells neu generiert wird. Nach jedem Generierungsvorgang müsste die manuelle Anpassung erneut erfolgen.

Diagramme Die Diagramme zu dieser alternativen Modellierung von Identifiern befinden sich im Anhang in Kapitel A.5.

Entscheidungsfindung Insgesamt musste die hier dargestellte alternative Modellierung von Identifier mit der bereits zuvor dargestellten Variante verglichen werden. Die bisherigen Erfahren der Palladio Gruppe mit dem Komponentenmodell zeigen, dass eine Festlegung auf GUIDs zur Erzeugung von Identifiern keine echte Einschränkung darstellt. Damit ist eine Austauschbarkeit der Erzeugung von IDs, die bereits über das Meta-Modell vorgesehen ist, nicht zwingend erforderlich.

Durch die ohnehin bestehende Trennung von Meta-Modell und Implementierung erfolgt bei keiner der Modellierungs-Alternativen eine Festlegung auf eine Implementierung zur Erzeugung von Identifiern. Über das UML2-Meta-Modell kann der konkrete Erzeugungsvorgang nicht beschrieben werden. Damit obliegt es in jedem Fall der Implementierung eines Modells, mit welchem Verfahren eindeutige IDs erzeugt werden.

Für eine Entscheidung zu Gunsten der in Kapitel 3.5.4.4 vorgestellten Alternative sprach die einfachere Modellierung, der geringere Aufwand durch die feste Implementierung, die fehlenden Einschränkungen, die durch die Festlegung auf die Verwendung von GUIDs entsteht, und die nach wie vor gegebene Austauschbarkeit der Realisierung von Identifiern über die Implementierung selbst. Die umgesetzte Implementierung wird im folgenden Kapitel 4.3.2 beschrieben.

4.3.2. Umgesetzte Implementierung für Identifier

Implementierung Für die Implementierung der Identifier wurde, wie bereits gesagt, die als einfacher betrachtete Variante gewählt, bei der über das Komponentenmodell keine Wahl der konkreten Realisierung geboten wird. Die Implementierung erfolgte sehr direkt durch Modifikation des Konstruktors der Klasse `IdentifierImpl`. Initial bei der Konstruktion neuer Instanzen wird der Wert des id-Attributs durch eine Bibliotheks-Klasse aus dem ECORE-Namespace gesetzt. Die Eigenschaften der verwendeten „UUID“ sind dabei ähnlich der GUID-Implementierung, wie sie beispielsweise von Marc A. Mnich [55] zur Verfügung steht. Die zufälligen Zeichenketten werden dabei auf Basis von MAC-Adressen, IP-Adressen, Zeitkomponenten und sequentiellen Zahlen generiert.

Der Konstruktor sieht im Quellcode von `IdentifierImpl.java` wie folgt aus:

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated not
 */
protected IdentifierImpl() {
  super();
  setId(org.eclipse.emf.ecore.util.EcoreUtil.generateUUID());
}
```

Da Identifier in einem eigenständigen UML2-Modell liegen, lässt sich der Export aus RSA gezielt für alle UML2-Modelle mit Ausnahme von Identifier initiieren. Damit wird zum einen die Modifikation der Identifier-ECORE-Datei („Id=true“ für das `id`-Attribut) nicht überschrieben. Da sich die ECORE-Datei für Identifier nicht ändert, führt die Generierung von Java-Code stets zum gleichen Ergebnis, *Merges* sind nicht notwendig, womit keine unerwünschten Änderungen an der Identifier-Implementierung auftreten. Durch Setzen des `generated not`-*Tags* bleiben die Änderungen am Konstruktor erhalten. Modifikationen am Modell „PalladioCM“ oder anderen Sub-Modellen des Komponentenmodells bleiben gleichwohl problemlos möglich und wirken sich nicht auf die Identifier aus.

Das `id`-Attribut wird somit für die Serialisierung von Modell-Instanzen verwendet. Dieses Attribut, gesetzt mit UUID-Werten, entspricht den Kriterien des Komponentenmodells, auch über Grenzen von Modellen hinweg eindeutig zu sein, womit sich verschiedene Modell-Instanzen einfach zusammenführen lassen.

Bewertung für den MDA-Ansatz Für die Idee des MDA-Ansatzes und den geplanten Prozess der Modellierung und Transformation bedeutet dies, dass Änderungen am UML2-Modell möglich sind und die Generierung von Java-Code mittels EMF ohne nachträglichen Aufwand für manuelle Anpassungen erfolgen kann. Zum jetzigen Zeitpunkt lassen sich damit noch keine Einschränkungen in der Produktivität durch die Verwendung des MDA-Ansatzes erkennen.

4.4. Testfall zur Evaluation

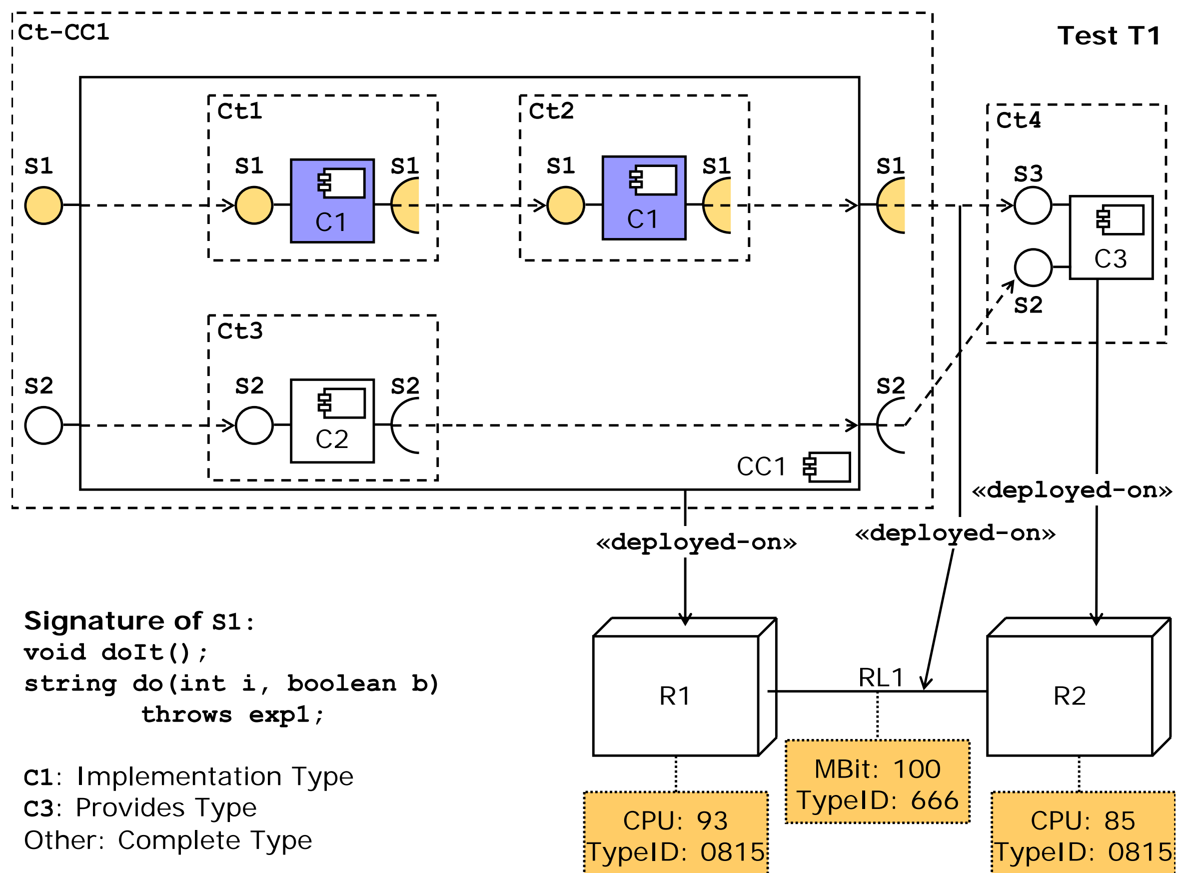


Abbildung 4.2.: Testfall für das Komponentenmodell

Idee Um die Fähigkeiten des mittels EMF generierten Modells (sowie die des generierten Editors) zu testen, diente ein Testfall, der alle wesentlichen Konzepte des Komponentenmodells enthält. Eine an ein UML2-Diagramm angelehnte Beschreibung des Testfalls ist in Abbildung 4.2 wiedergegeben. Außerdem stellt dieser Testfall eine Messkriterium für den mittels GMF generierten Editor (siehe Kapitel 5) dar. Je einfacher und vollständiger die in diesem Testfall enthaltenen Konzepte modellierbar sind, desto vollständiger ist die Umsetzung des Komponentenmodells mit EMF und GEF.

Der Testfall enthält keine Tests für Protokolle und SEFFs. Die Modellierung von Protokollen und SEFFs wird lediglich rudimentär abgedeckt, da sie nicht Kern des Interesses der Diplomarbeit sind. Wichtig war für die Diplomarbeit der Test, ob FSMs

und Petri-Netze sich dynamisch (durch Referenzierung der entsprechenden Modelle) zur Beschreibung von Protokollen und SEFFs verwenden ließen.

Testfall Der konstruierte Testfall („Test T1“) deckt die folgenden Konzepte des Komponentenmodells ab:

- *Basic Component.* C1, C2 und C3 stellen *Basic Components* dar.
- *Composite Component.* CC1 ist eine *Composite Component.*
- Schnittstelle. S1, S2 und S3 sind Schnittstellen.
- Rolle. Die gerade genannten Schnittstellen werden als *Provided* und *Requires Roles* verwendet. C3 bietet zwei Rollen an.
- Komponenten-Typ-Ebenen. C1 ist als *Implementation Type* realisiert, C3 ist ein *Provides Type*, die weiteren Komponenten sind *Complete Types*.
- Kontext. Alle Komponenten liegen als Kontext-Komponenten in unterschiedlichen Kontexten vor. C1 (Komponenten-Typ) wird in zwei Kontexten (Ct1 und Ct2) der gleichen Hierarchiestufe (CC1) verwendet.
- Konnektoren. Delegations-Konnektoren existieren für die angebotene und benötigte Seite von CC1. Assembly Konnektoren existieren zwischen S1 in Kontext Ct1 und S1 in Kontext Ct2 sowie zwischen S2 in Kontext Ct-CC1 und S2 in Kontext Ct4 (außerhalb eine *Composite Component*). Der Assembly Konnektor zwischen S1 von CC1 zu S3 von C3 verbindet zwei nicht identische Schnittstellen.
- Ressourcen. Die berechnenden Ressourcen R1 und R2 sind über die nicht-berechnende Ressource RL1 miteinander verbunden.
- Allokation. Auf R1 ist die Kontext-Komponente CC1 in Kontext Ct-CC1 allokiert, C3 in Kontext Ct4 auf R3 und die Assembly Konnektoren zwischen den genannten Komponenten auf RL1.
- Signatur. Auf der Schnittstelle S1 sind zwei Signaturen definiert. Eine der Signaturen hat mehrere Parameter sowie eine Exception.
- Annotationen. Zu R1, R2 und RL1 existieren Annotationen, wobei die Typen der Annotationen zu R1 und R2 dem gleichen Annotationstyp entsprechen.
- Identifier. Für erzeugte Entitäten musste automatisch ein Identifier (im Speziellen: UUID) generiert werden. Zudem musste die Referenzierung in der Serialisierung mittels der Identifier erfolgen.

In Abbildung 4.2 werden die Komponenten-Typen (Repository) nicht explizit aufgeführt. Alle Komponenten, die in der Abbildung zu sehen sind, entsprechen also Kontext-Komponenten zu den nicht aufgeführten Komponenten-Typen.

Protokolle und SEFFs Instanzen von Protokolle und SEFFs sollten durch dynamisches Referenzieren bzw. Importieren der entsprechenden Sub-Modelle über verschiedene Realisierungen modelliert werden können. Dabei sollten zu Protokollen und SEFFs mehrere unterschiedliche Realisierungsformen zur gleichen Zeit möglich sein. Zudem sollte die Verwendung referenzierter externer Modelle transparent erfolgen. Dies schließt die transparente Erzeugung neuer Protokoll- und SEFF-Entitäten mit ein.

Ergebnis der Durchführung des Testfalls Das mit EMF aus den UML2-Modellen erzeugte Java-Modell und der dazu generierte EMF-Editor deckten den Testfall vollständig ab. Auch war die dynamische Verwendung von FSMs und Petri-Netzen für Protokolle und SEFFs möglich.

5. Graphical Modeling Framework – Editor

Neben dem von EMF erzeugten Modell sollten weitere Transformationsschritte für das Komponentenmodell durchgeführt werden. Das Ziel dieses abschließenden Schritts der Diplomarbeit war die Erstellung eines graphischen Editors des durch UML2 modellierten und durch EMF erzeugten Meta-Modells auf Basis des GMF-Frameworks. GMF ist in der Lage, mit Hilfe von Transformationsanweisungen, *Mappings* sowie einem Daten-Modell einen graphischer Editor unter GEF zu erstellen. Im IBM Redbook [56] wird ein ähnlicher – jedoch manueller – Ansatz beschrieben, wie sich EMF und GEF gemeinsam in Eclipse-Projekten nutzen lassen.

GMF ist ein Eclipse-Projekt, dessen Entwicklungsarbeiten zu weiten Teilen parallel zur Diplomarbeit durchgeführt wurden. Stand zu Beginn der Diplomarbeit, eine frühe Entwicklerversion mit Milestone-Release 1.0 M3 zur Verfügung, war zum Ende der Diplomarbeit mit 1.0 M6 (RC0/RC1) bereits eine Version verfügbar, deren API laut Projektplan stabil sein sollte. Das finale Release war für den 23. Juni geplant – im Rahmen der Diplomarbeit konnten daher lediglich inkomplette Entwicklerversionen getestet werden.

Die durchgeführten Arbeiten zur Erstellung eines funktionsfähigen Editors mit GMF sind als prototypisch zu begreifen. Erst die finalen Versionen von GMF ließen erwarten, dass sich alle Funktionen ohne Fehler verwenden ließen. Daher wurde ein besonderes Augenmerk darauf gelegt, festzustellen, welche konzeptionellen und technischen Möglichkeiten mit der Verwendung von GMF zur Erstellung eines graphischen Editors für das Komponentenmodell verbunden waren. Ziel des Editors war damit nicht die Erstellung eines detaillierten Editors, sondern eine möglichst gute Abdeckung technischer und konzeptioneller Anforderungen an GMF. Wichtige Kriterien, die sich aus den Eigenschaften des Komponentenmodells ergaben, waren daher:

- Verwendung von GMF mit einem Gesamtmodell, dass in mehrere ECORE-Teil-Modelle zergliedert ist (PalladioCM, Annotation, Identifier, ...)

- Aufteilung des Editors in verschiedene Sichten – realisiert über mehreren Plugins (Sicht für Repository, Assembly und Allokation getrennt)

- Verlinken von Entitäten zwischen den Sichten. Daraus ergab sich die Anforderungen externe Entitäten („Shortcuts") zu unterstützen.

- Darstellung des Kontexts mit enthaltenen Kontext-Komponenten und Kontext-Rollen

- Darstellungsmöglichkeiten für *Composite Components* mit innneren Komponenten und Strukturen

- Erprobung eines handhabbaren Architektur-Design-Vorgangs mit Hilfe des Editors
- Unterstützung für flexible Plugins für Annotationen, SEFFs und Protokolle.
- Anpassung der graphischen Repräsentation an eigene Bedürfnisse.
- Ermittlung des Aufwands zur Erstellung neuer Versionen des Editors nach Änderungen am ECORE-Modell
- Ermittlung sinnvoller Vorgaben für die Mappings und Transformationen in GMF
- Feststellung von Problemgebieten bei der Arbeit mit GMF
- Editierbarkeit aller Attribute der Entitäten des Komponentenmodells
- Handhabbarkeit größerer Komponentenarchitekturen

5.1. Einführung in GMF

GMF stellt ein Plugin für Eclipse dar. Allein aus diesem Grund ist es nicht als eigenständiges Programm zu betrachten. Zusätzlich benötigt GMF die Fähigkeiten von EMF – dem Plugin, mit dem bereits in den vorigen Schritten der Transformation des Komponentenmodells gearbeitet wurde. Daneben setzt GMF voraus, dass GEF vorhanden ist. Die graphische Repräsentation erfolgt mit eben diesem Plugin. Darüber hinaus hängt GMF von EMFT – dem Technologiezweig von EMF – ab. Dieses besteht aus Sub-Projekten

- zur Validierung von Modellsinstanzen,
- zur Interpretation von OCL-Ausdrücken,
- zur Ausführung von *Queries* auf Modellinstanzen,
- und zur Durchführung von Transaktionen.

EMFT selbst ist eine Eclipse-Projekt, dass aus GMF hervorgangen ist.

GMF trennt bei der Definition von Transformationsanweisungen, Generierungsanweisungen und *Mappings* klar nach unterschiedlichen Belangen („Separation of Concerns“). Dies schlägt sich auch in den Serialisierungsartefakten GMFs nieder:

- Domänenmodell. GMF gewinnt Informationen zum Domänenmodell aus ECORE-Dateien, in denen das Domänenmodell liegt.
- genmodel. Diese von EMF erstellte Datei dient zur Gewinnung von Information über das von EMF generierte Java-Modell zum Domänenmodell.
- gmfgraph. In dieser Datei wird die graphische Repräsentation für Elemente der Zeichenfläche definiert. GMF unterteilt intern in die rein graphische Repräsentation (*Figure*) auf der Zeichenfläche und die zu zeichnenden Entitäten, die später im *Mapping* (gmfmap) referenziert werden. Auf diese Weise kann die gleiche grafische Repräsentation für mehrere Entitäten verwendet werden.

Für die graphische Repräsentation (*Figure*) kennt GMF vorgegebene Formen wie Rechtecke, abgerundete Rechtecke oder Linien. Zusätzlich werden selbst-definierte Formen erlaubt, um die graphische Erscheinungsweise den eigenen Bedürfnissen anzupassen (beispielsweise um *Interfaces* als Kreis darzustellen). Jede Standard-Form kann bereits durch GMF über Attribute wie Vorder- und Hintergrund-Farbe angepasst werden. Ebenso lassen sich Informationen zur Standard-Größe und weiteren Attributen hinterlegen.

Die zu zeichnenden Entitäten unterteilt GMF in *Node*, *Connection*, *Compartment* und *Label*. *Nodes* sind dabei Knoten-Elemente, die über *Connections* verbunden werden können. *Label* sind Beschriftungen im Diagramm und *Compartments* definieren schließlich Knoten, die innerhalb von anderen Knoten liegen können. Jede zu zeichnende Entität stellt mindestens ein Tupel aus *Figure* und einem eigenen Namen dar. Ergänzend sind weitere Attribute, abhängig von der Art der zu zeichnenden Entität, zu definieren. Auch wenn im Regelfall jede zu zeichnende Entität einer Entität des Domänenmodells entsprechen dürfte, erfolgt über gmfgraph noch keine Abbildung auf Entitäten des Domänenmodells.

- gmftool. Diese Datei enthält Informationen über Elemente (*Tools*) der *Toolbar*. Jedes *Tool* besteht neben einer *Tool*-Beschriftung und einem *Tooltip* aus der Angabe zweier *Icons* für verschiedene Auflösungen, die dann als Symbole in der *Toolbar* erscheinen.

- gmfmap. Das *Mapping* für GMF wird in der gmfmap-Datei realisiert. Über das *Mapping* werden graphische Repräsentation, *Tool*-Definition und Domänenmodell miteinander verknüpft. Daher darf dieses *Mapping* als Kern der Erstellung von Editoren mittels GMF gesehen werden.

  Im *Mapping* wird auf oberster Eben zwischen jenen Elementen unterschieden, die als Knoten direkt auf der Zeichenfläche erscheinen sollen (*Top Node Reference*) und jenen Elementen, die als *Link* (also Verbindung zwischen Knoten) realisiert werden sollen. *Nodes* verknüpfen dabei intern einen Knoten aus der graphischen Repräsentation, ein Erzeugungs-*Tool* und eine Entität (`EClass`) aus dem Domänenmodell. *Links* verknüpfen ebenfalls eine graphische Repräsentation, ein Erzeugungs-*Tool* und (bei Assoziations-Klassen) eine Entität des Domänenmodells miteinander. Zusätzlich werden die zu setzenden Attribute der Entität des Domänenmodells, die Quelle und Ziel eines *Links* darstellen, spezifiziert. Sowohl *Nodes* als auch *Links* geben als *Containment Feature* an, als Attribut welcher Domänenmodell-Entität sie erzeugt werden (im Falle des Komponentenmodells werden beispielsweise nahezu alle Entitäten von der *Factory* erzeugt).

  Im Mapping werden auch *Compartments* spezifiziert. Siehe hierzu auch Kapitel 5.3.3.

- gmfgen. Dieses Artefakt enthält die Generierungsanweisungen von GMF zur Erstellung des Diagramm-Plugins und -Quelltextes. Grundsätzlich wird zu jedem Knoten aus dem *Mapping*, allen *Links* und allen *Compartments* ein Ast in gmfgen angelegt. Jeder Ast entspricht dabei mehreren Klassen. Für jeden Ast, der von GMF angelegt wurde, wird eine Klasse zur Abbildung der Bearbeitungsfunktion für Entitäten angelegt, eine Klasse die *Policies* für Bearbeitungsfunktionen entspricht und eine *Factory* zur Erzeugung von graphischen Elementen. Für *Nodes*

werden noch zwei zusätzliche Klassen für die graphische Repräsentation und für weitere Bearbeitungs-*Policies* angelegt.

5.2. Erstellungsprozess

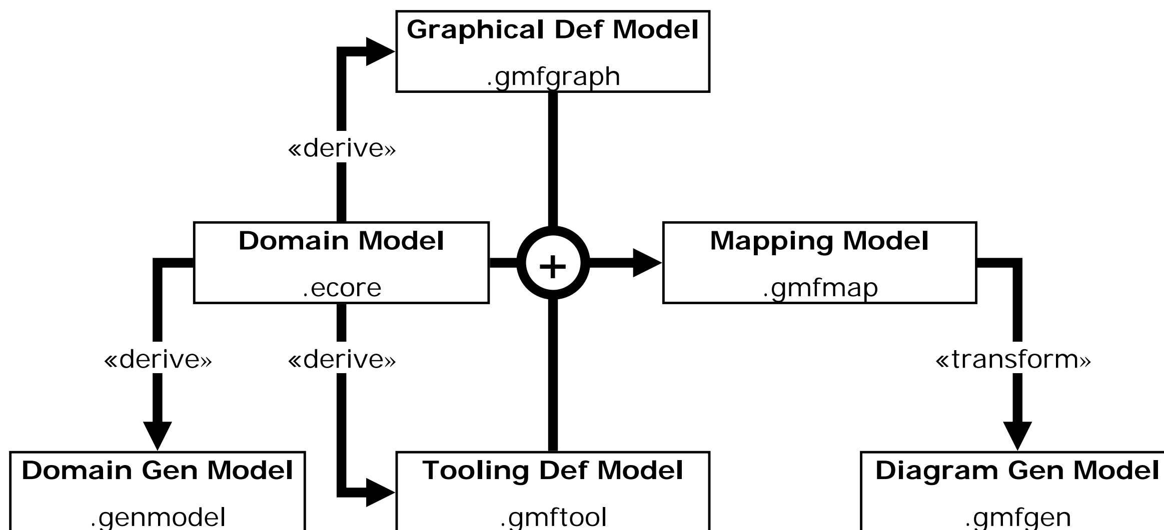


Abbildung 5.1.: Übersicht über den Verarbeitungsprozess von GMF (nach dem *Dashboard* des GMF-Plugins)

Nachdem im vergangenen Kapitel die Artefakte von GMF dargestellt wurden, wird in Abbildung 5.1 der Zusammenhang zwischen den Artefakten als Erzeugungsprozess skizziert. Aus dem Domänenmodell (*Domain Model*) wird das Domänen-Generierungsmodell (*Domain Gen Model*) abgeleitet. Dieser Schritt erfolgt durch EMF, nicht durch GMF, ist aber dennoch Grundlage zur Erstellung des GMF-Editors, da hiermit der Java-Quellcode für das Domänenmodell erstellt wird.

Zu den Entitäten des Domänenmodells werden (üblicherweise über *Wizards* von GMF) die graphische Definition (*Grahical Def Model*) und die *Tool*-Definition generiert. Nachdem eine manuelle Anpassung dieser vor-generierten Modelle durchgeführt wurde, erfolgt die Erstellung des *Mappings* (*Mapping Model*). Hierin erfolgt die Kombinierung von gmfgraph, ecore und gmftool.

Aus dem *Mapping* wird schließlich in einem Transformationsschritt ein Generierungsmodell für das Diagramm-Plugin (*Diagramm Gen Model*) erstellt. Am genmodel vorgenommene manuelle Änderungen fließen schließlich in den Erzeugungsprozess für Java-Quellcode des generierten Editor-Plugins ein.

5.3. Realisierung

An dieser Stelle sollen grundsätzliche Ideen zur Umsetzung des Editors mit GMF dargestellt werden. Eine detailliertere Anleitung zur Erstellung eines Editors für das Komponentenmodell befindet sich in Kapitel A.3.

5.3.1. Graphische Repräsentation

Zur Erstellung des Editors mittels GMF musste zunächst eine Abbildung der Strukturen des Komponentenmodells auf die graphischen Konstrukte GMFs erdacht werden. Als Darstellungsformen stehen grundsätzlich (wie bereits angesprochen) drei Varianten zur Auswahl:

- *Nodes*. Knoten
- *Links*. Verbinden je zwei Knoten miteinader
- *Compartments*. Knoten, die in Knoten enthalten sind

Die Umsetzung der Repräsentation erfolgte mittels gmfgraph. Grundsätzlich wurden alle Entitäten des Komponentenmodells als Knoten realisiert. Ausgenommen davon waren: *Provided Role*, *Required Role*, *Assembly Connector*, *Provided Delegation Connector*, *Required Delegation Connector* und *Non-Calculating Resource*. Ergänzend als *Containment* wurden realisiert:

- Komponenten im Inneren von *Composite Components*
- In der Allokationssicht: Kontext-Komponenten in berechnenden Ressourcen zur Visualisierung der *allocated-on*-Relation.

5.3.2. Tool-Definition

Grundsätzlich entsprach jeder Entität des Komponentenmodells eine eigene *Tool*-Definition in gmftool. Da GMF nicht zwischen *Tools* zur Erzeugung von *Links* und *Nodes* unterscheidet, mussten keine weiteren Differenzierungen vorgenommen werden.

5.3.3. Mapping-Definition

Grundsätzlich wurde zu jeder Entität des Komponentenmodells (Domänenmodell) entsprechend der Ideen zur Unterteilung in der graphischen Repräsentation ein *Node*, *Compartment* bzw. *Link* angelegt, bei dem das *Tool* der dazugehörigen Entität sowie die für die jeweilige Entität erstellte graphische Repräsentation verwendet wurde. Als Modellierungselement wurde die Entität aus dem ECORE-Domänenmodell gewählt.

Auf diese Weise erhielten alle in Kapitel 5.3.1 genannten *Links* eine Assoziationsklasse. Das Ziehen einer Linie als graphische Repräsentation im Editor resultierte in einer neuen Instanz der dazugehörigen Klasse aus dem ECORE-Modell, wobei die mit der Linie assoziierten Knoten direkt als Attribute der Assoziationsklasse gesetzt wurde.

Eine Instanz einer *Provided Role*, die als *Link* realisiert wurde, konnte also durch Ziehen auf der Zeichenfläche zwischen Komponenten und Schnittstelle erzeugt werden. Dabei wurden die angebotene Schnittstelle und die anbietende Komponente als Attribute der Instanz automatisch gesetzt.

5.3.4. Sichten

GMF erlaubt grundsätzlich verschiedene Sichten auf das gleiche Domänenmodell. Dazu werden mehrere Diagramm-*Plugins* zum gleichen Domänenmodell erzeugt. Zusammen

bilden diese Diagramm-*Plugins* einen GMF-Editor für das Komponentenmodell. Jedes Diagramm-*Plugin* ist in der Lage, seine eigene graphische Repräsentation sowie sein eigenes Bearbeitungsverhalten für die Entitäten eines Domänenmodells zu definieren. Die Sichten sind – bis auf ein gemeinsames Datenmodell – vollkommen unabhängig voneinander. Werden die auf diese Weise erzeugten Diagramm-*Plugins* im gleichen Eclipse-*Workspace* verwendet und wird zur Serialisierung des Domänenmodells die gleiche Ressource gewählt, lässt sich dieselbe Instanz des Domänenmodells in allen Sichten bearbeiten.

Für den GMF-Editor für das Komponentenmodell war eine Aufteilung auf drei Sichten geplant, die jeweils spezielle Aspekte des Komponentenmodells hervorheben.

5.3.4.1. Repository Sicht

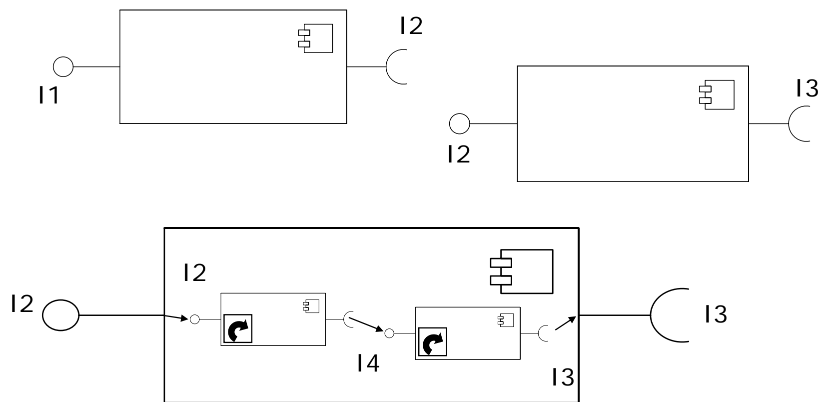


Abbildung 5.2.: *Repository* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8])

In der Repository Sicht (vgl. Abbildung 5.2) werden lediglich Komponenten-Typen als eine ungeordnete Menge dargestellt. Die primär in dieser Sicht verwendeten Entitäten sind Schnittstellen, Rollen und Komponenten-Typen. Die auf diese Weise definierbaren Komponenten-Typen werden im Falle von *Basic Components* um SEFFs ergänzt. Eine Sicht auf SEFFs – gleich welcher konkreten Ausprägung – ist jedoch nicht Teil dieser Sicht, sie würden von einem eigenständigen Editor bearbeitbar.

Da im Inneren von *Composite Components* nur Kontext-Komponenten liegen können, handelt es sich hierbei bereits um Kontext-Instanzen von Komponenten-Typen, die bereits im Repository definiert wurden. In der Abbildung wird dies durch *Shortcut*-Symbole angedeutet. Auf eine exlipzite Darstellung des Kontexts wird dabei verzichtet. Dennoch bleibt die Darstellung eindeutig. Jegliche Verwendungen von Komponenten-Typen innerhalb von *Composite Components* liegen implizit in einem Kontext.

Die inneren Strukturen von *Composite Components* werden mit nur genau einer Hierarchie-Stufe dargestellt. Da jede *Composite Component* explizit mit genau einer Hierarchie-Stufe dargestellt wird, lässt sich die Gesamthierarchie in der Assembly-Sicht rekonstruieren. Die Verwendung von Kontext-Komponenten folgt dabei dem *Black-Box*-Gedanken – innere Strukturen von Kontext-Komponenten werden abstrahiert.

5.3.4.2. Assembly Sicht

In der Assembly Sicht (vgl. Abbildung 5.3) können Komponentenarchitekturen modelliert werden. Als Entitäten werden dazu Kontext-Komponenten, Kontext-Rollen und Assembly Konnektoren dargestellt. Das Innere von Kontext-Komponenten wird dabei vollständig ausgeblendet (*Black-Box*-Sicht). Der Verweis auf Komponenten-Typen zu Kontext-Komponenten erfolgt dabei (analog zum Verhalten in *Composite Components*) über *Shortcuts*. Aus den gleichen Gründen wie bei der Betrachtung von *Composite Components* wird der Kontext nicht explizit dargestellt. Da jegliche Entitäten Entitäten des Kontexts sind, ist keine Unterscheidung zwischen Komponenten-Typen und Kontext-Komponenten notwendig.

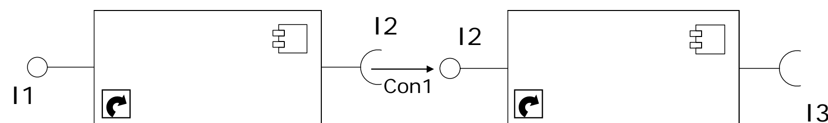


Abbildung 5.3.: *Assembly* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8])

5.3.4.3. Allokations-Sicht

Über die Allokations-Sicht (siehe Abbildung 5.4) erfolgt eine (nahezu) komplette Abstraktion von den Strukturen der Komponentenarchitektur, so wie sie in der Assembly Sicht definiert wurden. Welche Kontext-Komponente über welche Assembly Konnektoren mit einer anderen Kontext-Komponente verbunden ist, wird vollständig ausgeblendet. Dafür werden in dieser Sicht zwei Aspekte der Allokation zusammengeführt:

1. Die Verbindungsstrukturen zwischen berechnenden und nicht-berechnenden Ressourcen werden definiert.

2. Es erfolgt eine Zuordnung von Kontext-Komponenten zu berechnenden Ressourcen und von Assembly Konnektoren zu nicht-nerechnenden Ressourcen.

Zur Visualisierung wird die Zuordnung, wie in der UML2 Deployment-Sicht, über eine *Containment*-Beziehung zwischen Box-Symbol und Kontext-Komponenten mit einem *Shortcut*-Symbol vorgenommen.

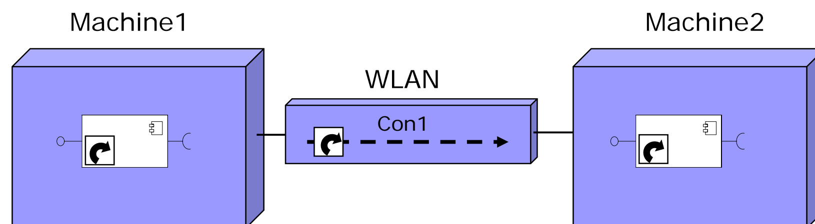


Abbildung 5.4.: *Allocation* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8])

5.3.4.4. Umsetzung

Die in den Kapiteln 5.3.4.1 bis 5.3.4.3 aufgeführte Visualisierung stellt lediglich ein mögliches Konzept zur Umsetzung dar. Eine tatsächlich derart detaillierte Ausgestaltung des GMF-Editors war im Rahmen der Diplomarbeit nicht vorgesehen. Die Umsetzung im Rahmen der Diplomarbeit zielte auf die Erprobung der dafür notwendigen Konzepte.

Auf Grund eines *Bugs*[1], der bis zur in der Diplomarbeit verwendeten GMF Version 1.0M6 existierte, ließ sich eine vollständige Umsetzung der Aufteilung auf die oben beschriebenen Sichten nicht vornehmen. Auf Grund des *Bugs* müssen die Mengen in einer Sicht modellierbarer Entitäten zwischen den gleichzeitig in einer Eclipse-Instanz laufenden Diagramm-*Plugins* disjunkt sein. Sollte also die Entität `ContextComponent` sowohl in der Repository und Assembly Sicht verwendet werden, war dies auf Grund des *Bugs* nicht möglich.

Trotz der Einschränkungen in der Aufteilung auf mehrere Sichten, wurden so viele weitere Konzepte wie möglich mit der vorliegenden Version von GMF getestet.

5.3.5. Umgesetzte Konzepte

Label Alle graphischen Elemente (`Node`, `Link`, `Compartment`) tragen zur Beschriftung *Label*. Das *Label Mapping* zeigt dabei den Namen der Entitäten und die ID lesbar an und erlaubt die Manipulation des Namens. Ergänzend wird der Typ der Entität als Beschriftung angezeigt, damit auch graphisch identisch dargestellte Entitäten unterscheidbar bleiben.

Custom Figures Exemplarisch wurde die Zeichenform des *Interfaces* verändert. Um eine an das UML2 Komponentendiagramm angelehnte Darstellung zu erreichen, wurde eine Kreisform aus der Bibliothek von GEF zur Visualisierung verwendet. Auf diese Weise konnte erfolgreich die Anpassung der Darstellung von Knoten erprobt werden.

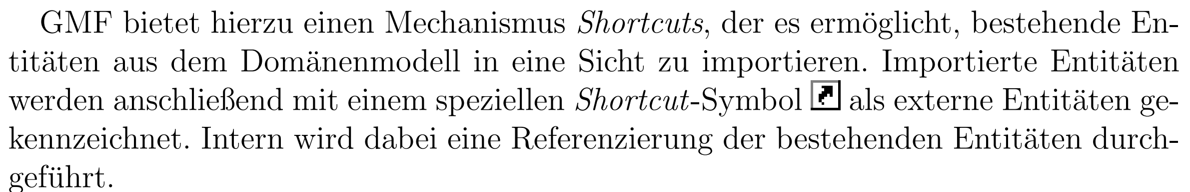
In vereinfachter Weise ließen sich die Darstellungsformen von `CMEnvironment` und `Annotation` anpassen. Hier wurde auf die Standard-Mittel von GMF zurückgegriffen: Genutzt wurden Veränderungen der Farben für Vorder- und Hintergrund sowie die Wahl anderer Grundformen.

Benutzerdefinierte Pfeil-Formen Der Anleitung aus dem GMF Tutorial [35] folgend, ließen sich beliebige Pfeilformen erzeugen. Für Assembly Konnektoren wurde ein offener Pfeil gewählt. Ergänzend wurde der Linientyp auf eine gestrichelte Form gestellt. Damit waren UML2-konforme Darstellungen des Konnektors möglich. Eine selbst-definierte Pfeil-Form wurde ebenfalls zur Visualisierung der benötigten Schnittstelle als „Halbkreis“ gewählt.

Erstellung von Links auf externe Entitäten Durch die Nutzung verschiedener Sichten auf das Komponentenmodells erwuchs die Notwendigkeit, bestehende Entitäten in

[1] Bugzilla Bug #136760: https://bugs.eclipse.org/bugs/show_bug.cgi?id=136760;
Diskussion des Bugs in der Mailingsliste:
http://www.eclipse.org/newsportal/article.php?id=2269&group=eclipse.technology.gmf#2269

unterschiedlichen Sichten darzustellen. Soll etwa die Allokation von Kontext-Komponenten auf berechnende Ressourcen in der Allokations-Sicht im GMF-Editor modelliert werden, müssen die in der Assembly-Sicht definierten (bestehenden) Kontext-Komponenten mit berechnenden Ressourcen verknüpft werden.

GMF bietet hierzu einen Mechanismus *Shortcuts*, der es ermöglicht, bestehende Entitäten aus dem Domänenmodell in eine Sicht zu importieren. Importierte Entitäten werden anschließend mit einem speziellen *Shortcut*-Symbol  als externe Entitäten gekennzeichnet. Intern wird dabei eine Referenzierung der bestehenden Entitäten durchgeführt.

Compartment Um zusammengesetzte Strukturen – etwa für das Innere von *Composite Components* – darstellen zu können, bietet GMF die Möglichkeit *Compartments* zu definieren. Wie bereits beschrieben wurde, wirkt sich dieses auch auf die Definition graphischer Elemente (gmfgraph) aus. Für die graphische Repräsentation werden dabei die zu verwendenden *Figures* festgelegt.

Zusätzlich muss das *Compartment* im *Mapping* (gmfmap) definiert werden. Über eine *Child Reference* wird für alle Entitäten, die im Inneren eines Knoten vorkommen können sollen, festgelegt, auf welche Entitäten des Domänenmodells innere Knoten abgebildet werden und welches ECORE-Attribut für die enthaltende Entität gesetzt werden muss. Ein Eintrag *Compartment Mapping* übernimmt dabei die Verknüpfung der graphischen Repräsentation und der *Child References* im *Mapping*.

Compartments lassen sich zusätzlich über Eingriffe in das Generierungsmodell für das Diagramm-Plugin (gmfgen) steuern. Insbesondere das Attribut *List Layout* auf dem Modellelement *Gen Compartment* aus gmfgen ist hier hervorzuheben, da über dieses Attribut gesteuert wird, ob enthaltene Elemente lediglich als Listen von *Labeln* dargestellt werden oder als frei bewegliche *Figures*, die sich über *Links* mit weiteren *Figures* verbinden lassen. Zur graphischen Darstellung von Beziehungen über *Links* zwischen enthaltenen Elementen muss das *List Layout* daher zwingend deaktiviert werden.

Signatur-Editor Um ein einfaches Bearbeiten von Signaturen zu ermöglichen, wäre ein graphischer Editor unangemessen. Die Standard-Darstellungsform für Signaturen besteht in einer textuellen Darstellung. Bei dieser Form der Darstellung ist vor allem die Reihenfolge der Signaturen einfacher zu erfassen. Ein entsprechender graphischer Editor müsste zur Darstellung einer Reihenfolge gesonderte Konstrukte beinhalten. Aus diesem Grund wurde kein graphischer Editor für Signaturen erstellt.

Gleichwohl gibt es in GMF die Möglichkeit manuell Editoren für das *Property Sheet* von Entitäten zu erstellen. Mit der angeratenen Vorgehensweise würde der durch GMF generierte Editor für das Signatur-Attribut der *Interfaces* modifiziert und durch einen selbst geschriebenen Signatur-Editor ersetzt. Dieses Vorgehen wird auch in der GMF Newsgroup vorgeschlagen, um komplexere Editoren für Attribute von ECORE-Entitäten zu erzeugen. Dies wurde im Rahmen der Arbeit jedoch nicht evaluiert.

Editierbarkeit aller Attribute Um das gesamte Komponentenmodell sinnvoll bearbeiten zu können, ist es notwendig, dass tatsächlich alle im Meta-Modell definierten Attribute, Assoziationen, Aggregationen und Kompositionen bearbeitet werden können. Handelt es sich um Attribute, die aus Standard-Datentypen aus ECORE

(`EInt`, `EString`, etc.) bestehen, können diese direkt im *Property Sheet* bearbeitet werden.

Komplexe Attribute (wie die vorangehend genannten Signaturen) können über selbst geschriebene Editoren oder aber über einen (eigenen) graphischen GMF-Editor bearbeitet werden. Dabei können Assoziationen und Aggregationen des Meta-Modells den Instanzen über *Links* zugewiesen werden. Besteht etwa eine `1:n`-Beziehung zwischen *Interface* und Signaturen, können diese über `n` *Links* zwischen einer *Interface*-Instanz und einer Signatur-Instanz ausgedrückt werden.

Sollen Beziehungen zwischen Instanzen von Entitäten des Meta-Modells nicht graphisch über *Links* modelliert werden können, bietet GMF die Möglichkeit auf die Fähigkeiten des generierten EMF-Editors (vgl. Kapitel 4.2.1.3) zurückzugreifen. Alle Attribute zu anderen Entitäten des Meta-Modells (inklusive Attributen, die im Meta-Modell über Assoziationen ausgedrückt sind), für die durch GMF keine Möglichkeiten zur graphischen Modellierung erzeugt werden, können über Drop-Down-Menüs mit bereits erzeugten Instanzen von Entitäten belegt werden (neue Instanzen können indes nicht erzeugt werden; siehe folgender Abschnitt).

Erzeugung von Instanzen GMF kennt vier Wege, auf denen neue Instanzen von Entitäten des Meta-Modells erzeugt werden können:

1. Der Standard-Weg um neue Instanzen von Entitäten zu erzeugen, ist die Verwendung von *Creation Tools*, die in der *Toolbar* aufgeführt werden. Im Falle des Komponentenmodells sind damit alle *First Class Entities* des Komponentenmodells erzeugbar. Zusätzlich lassen sich Annotationen und alle Elemente des Kontexts über Tools direkt erzeugen.

   GMF erlaubt die Erzeugung neuer Entitäten auf der Zeichenfläche mittels *Tools* nur für solche, die von einer zentralen Entität erzeugt (im Falle des Komponentenmodells der `Factory`) und als Komposition geführt werden. Daher wurde gegenüber dem für das reine EMF erstellten Meta-Modell zusätzlich eine Kompositionsbeziehung zwischen *Factory* und Annotation sowie Kontext-Komponenten und den Kontext-Rollen im UML2-Modell eingeführt.

2. Compartments bieten eine weitere Möglichkeit, um Instanzen von Entitäten zu erzeugen. Dabei ist die Erzeugung auf das „Innere“ der umgebenden Instanz beschränkt. Analog zur Logik zur Erzeugung von Instanzen von Entitäten auf der Zeichenfläche, ist die Erzeugung von Entitäten nur zugelassen, wenn die umgebende Entität im Meta-Modell die innere Entität per Komposition (im UML2-Modell) führt.

3. Auch mittels *Links* lassen sich in bestimmten Fällen Instanzen von Entitäten erzeugen. Werden *Links* (als Assoziationsklassen) zu Entitäten definiert, wird mit dem Erzeugen des *Links* eine Instanz der entsprechenden Entität erzeugt.

4. Eine indirekte Form der Erzeugung von Entitäten besteht über die Nutzung von *Shortcuts*. Indem Entitäten in externen Diagrammen angelegt werden, können diese anschließend in ein anderes Diagramm importiert werden. Hierbei handelt es sich genau genommen um keine Erzeugung von Instanzen von Entitäten. Gleichwohl bleibt die Erzeugung aus Sicht des importierenden Diagramm-Plugins verborgen.

Insbesondere die Bindung des Erzeugungsprozesses neuer Entitäten an die Existenz einer Kompositions-Beziehung zwischen Entitäten erschwert die freie Gestaltung des Editors unter GMF. Im Falle von Annotationen beispielweise können diese im Verständnis des Komponentenmodells nur abhängig von einer Entität (`Entity`) existieren. In der UML2-Modellierung wurde dies durch eine Kompositionsbeziehung zwischen `Entity` und `Annotation` ausgedrückt. Im GMF-Editor war es gewünscht, Annotationen – wie auch in UML-Diagrammen üblich – als eigene *Figures* darstellen zu können und dann über *Links* mit `Entity`-Instanzen verbinden zu können. Da die Annotation-*Figure* in GMF jedoch auf der Zeichenfläche erstellt werden soll, war eine Kompositionsbeziehung (wie bereits oben beschrieben) zwischen *Factory* und Annotation eigens für GMF zu erstellen.

An dieser Stelle zeigt sich eine konzeptionelle Einschränkung von GMF. Die in einem GMF-Editor möglichen Erzeugungsprozesse richten sich nach den Vorgaben des Domänenmodells. Zunächst einmal wird auf diese Weise garantiert, dass nur solche Instanzen modelliert werden können, die eine gültige Ausprägung des Domänenmodells sind. Wie für den Fall der Annotation wäre es jedoch wünschenswert, dass abweichende Erzeugungsprozesse ebenfalls möglich sind, solange diese letztlich zu validen Konstrukten des Domänenmodells führen. Unvollständige Modellierungskonstrukte könnten durch eine Validierung erkannt werden. Damit würde eine Trennung zwischen den Erzeugungsprozessen im Domänenmodell und in GMF-Editoren zu Gunsten größerer Flexibilität in der Modellierung eingeführt.

Handhabbarkeit größerer Komponentenarchitekturen Über die von GMF in Kombination mit GEF gebotenen Zoom-Funktionen, können auch größere Architekturen übersichtlich bleiben. Ergänzend lassen sich zu einer ECORE-Modell-Instanz mit dem gleichen Plugin (also auch der gleichen Sichtweise) mehrere Diagramme erzeugen. Dazu wird eine neue Diagramm-Instanz zu einer bestehenden ECORE-Modell-Instanz angelegt. Die Verlinkung zwischen den Diagrammen erfolgt über den bereits eingeführten *Shortcuts*-Mechanismus. Verschiedene Aspekte der gleichen Komponentenarchitektur lassen sich damit auf unterschiedliche Diagramme (der gleichen Sicht) verteilen.

5.3.6. Probleme und Einschränkungen

Die in diesem Kapitel genannten Probleme beziehen sich grundsätzlich, sofern nicht anders vermerkt, auf die Version 1.0M6 von GMF, inklusive der von GMF benötigten Programmpakete von EMFT, EMF und EclipseSDK.

Kontext Um beispielweise den Kontext als *Figure* mit darin enthaltenen Kontext-Komponenten und Kontext-Rollen als *Compartment* darstellen zu können, wäre eine Funktionalität wünschenswert, die beim Erzeugen einer neuen Kontext-Komponente in einem Kontext gleichzeitig alle vom Komponenten-Typ angebotenen und benötigten Rollen darstellt. Da die Zuordnung von Kontext-Rollen zu Kontext-Komponenten bereits durch den Komponenten-Typ erfolgt, müssten stets alle vom Komponenten-Typ definierten Rollen ebenfalls im Kontext erscheinen.

In Abbildung 5.5 wird das beschriebene Problem dargestellt. Nach der Erstellung einer Kontext-Komponente zu einem bestehenden Komponenten-Typ werden die für den Komponenten-Typ vorhandenen Rollen nicht für die Kontext-Komponente angelegt.

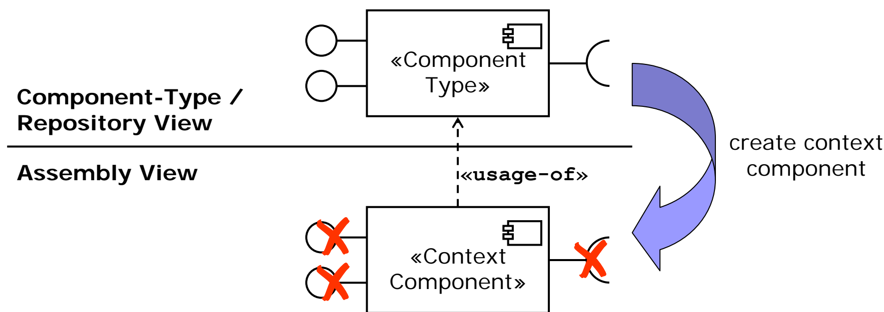


Abbildung 5.5.: Erstellung einer Kontext-Instanz eines Komponenten-Typs

GMF bietet den Mechanismus *Feature Sequence Initializer*. Die Idee dieses Initialisierungshelfers bezieht sich jedoch nicht auf die Erzeugung untereinander abhängiger *Figures*, sondern auf die Initialisierung von Attributen einer neu erzeugten Entität. (EMF erlaubt die Definition von *Default Values* für Attribute, jedoch handelt es sich hierbei nicht um Werte, die tatsächlich für eine Attribut gesetzt werden, sondern um eine Standard-Interpretation des `null`-Wertes. GMF übernimmt mit *Feature Sequence Initializer* die tatsächliche Belegung von Attributen mit Werten im Zuge des Erzeugungsprozesses von Instanzen von Entitäten.)

Shortcuts GMF erlaubt *Shortcuts* nur für Entitäten, für die eine graphische Repräsentation inklusive *Mapping* im importierenden Diagramm-*Plugin* vorhanden ist. Hierbei handelt es sich um eine gewollte Einschränkung durch GMF. Ein Mechanismus, der es erlaubt, dynamisch beliebige Entitäten zu importieren, wäre nicht sinnvoll, da für diese Entitäten im importierenden Diagramm-*Plugin* kein *Mapping* definiert wäre.

Die von GMF umgesetzte Unterstüzung von *Shortcuts* lässt dennoch viele Freiheitsgrade, denn die Darstellung per *Shortcut* importierter Entitäten ist rein vom importierenden Diagramm-*Plugin* abhängig. Damit sind die Sichten auf ein Domänenmodell in ihrer Darstellung vollständig frei. Sind in einem Diagramm-*Plugin* Assoziationsklassen in Form von *Links* realisiert, können diese beispielsweise in einer anderen Sicht als *Nodes* dargestellt werden.

Automatisches Mapping Das automatische *Mapping* zwischen Entitäten und *Tools* schlägt fehl. Obwohl GMF in der Lage ist, sowohl *Tools* als auch graphische Repräsentationen, *Nodes* und *Links* zu Entitäten automatisch zu erzeugen, schlägt die Zuordnung zwischen Entitäten und *Tools* (automatisiert) fehl. Dies führt dazu, dass beispielsweise ein mit *Composite Component* beschriftetes *Tool Interfaces* erzeugt. Eine Änderung dieser Einschränkung des automatisierten *Mappings* bleibt für zukünftige *Releases* zu erwarten, da das automatische *Mapping* zwischen Entität und graphischer Repräsentation bereits funktioniert. Eine manuelle Neuzuordnung der *Tools* funktioniert in jedem Fall (allerdings nicht über den *Mapping Wizard*, sondern nur über die Baumansicht von gmfmap).

Referenzierungsmechanismen in GMF: Tools *Tools* werden in gmfmap über „absolute“ DOM-Ausdrücke referenziert (etwa

`href=''PalladioCM.gmftool#//@palette/@tools.0/@tools.6''`). Im Standard-Fall des Hinzufügens neuer *Tools* in gmftool ist diese Form der Referenzierung unkritisch, da neue *Tools* am Ende der Liste angehängt werden. Wird jedoch ein *Tool* aus der Mitte der Liste entfernt, werden alle Referenzen auf nachfolgende *Tools* falsch, da Referenzen nicht automatisch durch GMF konsistent gehalten werden. Im *Worst-Case* (Löschen des ersten *Tools*) müssen daraufhin alle *Tool*-Referenzen in gmfmap neu gesetzt werden.

Referenzierungsmechanismen in GMF: Sonstige Die Referenzierung anderer Elemente in gmfmap, gmftool und gmfgraph erfolgt (mit Ausnahme der *Tools* in gmfmap) über die Namen der zu referenzierenden Elemente. Das bedeutet, dass Figures, Nodes ö. ä. immer einen eindeutigen Namen haben müssen – auch über den Element-Typ hinweg. Werden Namen unbeabsichtigt mehrfach verwendet, werden die ECORE-Elemente verwendet, die durch den Referenzierungsmechanismus als erstes gefunden werden. Elemente in den ECORE-Dateien von GMF tragen außer den Namen kein identifizierendes Merkmal. Eine Einführung von eindeutigen (generierten) IDs, wie sie im Komponentenmodell verwendet werden, wäre hier wünschenswert. Die Referenzierung anderer ECORE-Elemente könnte dann eindeutig über die ID erfolgen.

Der Referenzierungsmechanismus des Komponentenmodells, der bereits über IDs arbeitet, wird durch GMF nicht modifiziert. Die Eindeutigkeit der Referenzierung bei gleichem Namen von Instanzen von Entitäten des Komponentenmodells bleibt damit erhalten.

Referenzierung von ECORE-Elementen in externen ECORE-Dateien Werden externe Ressourcen unter GMF referenziert, erfolgt dies im Regelfall (eine Ausnahme bilden wiederum die *Tools*) über den qualifizierten Namen der ECORE-Elemente (etwa `href=''PalladioCM.ecore#//Entity/entityName''`). Eine Referenzierung von ECORE-Dateien aus anderen Eclipse-Projekten ist dabei nicht vorgesehen.

Sollen mehrere Diagramm-*Plugins* zum gleichen Domänenmodell erzeugt werden, müssen deshalb *Kopien* des Domänenmodells in jedem Diagramm-*Plugin*-Projekt erzeugt werden. Die Erzeugung mehrerer *Plugins* aus einem einzelnen Projekt heraus ist von GMF nicht vorgesehen. Unter anderem schränkt GMF die Benennung von gmfmap, gmftool und gmfgraph ein. Die Präfixe dieser Dateien müssen mit dem Namen der ECORE-Datei des Domänenmodells übereinstimmen, die Postfixe sind durch GMF fest vorgegeben. Damit lassen sich in einem Projekt zunächst keine zusätzlichen GMF-Projekte realisieren, weil die Dateien dieser Projekte die bereits belegten Dateinamen verwenden müssten.

Eine Möglichkeit über mehrer Diagramm-*Plugins* hinweg das gleiche Domänenmodell zu referenzieren, wäre wünschenswert, um eine unnötige Duplikation des Domänenmodells vermeiden zu können.

Unterstützung externer Plugins für Annotationen, SEFFs und Protokolle Die Unterstützung von externen Plugins zur Modellierung von Annotationen, SEFFs und Protokollen sowie die Assoziation dieser Entitäten zu bestehenden Entitäten einer Instanz des Komponentenmodells soll am Beispiel von Annotationen veranschaulicht werden. Annotationen sollen allen `Entities` des Komponentenmodells zugeordnet werden können. Dabei sind sie bezüglich ihres Typs nicht vorhersehbar. Auch komplexe Annotationen sind möglich. Aus diesem Grund sind eigene Plugins notwendig, die jeweils

eine begrenzte Menge von Annotations-Typen bearbeiten können. Plugins für Annotationen müssen dabei dynamisch geladen werden können.

Um die genannten Anforderungen an Annotationen erfüllen zu können, wurden wiederum *Shortcuts* verwendet. Soll ein spezieller Annotations-Typ in einer Instanz des Komponentenmodells unterstützt werden, so ist zunächst das Annotations-Plugin in den Eclipse *Workspace* zu laden. Soll eine Entität des Komponentenmodells annotiert werden, so wird zunächst eine Annotation im Annotations-Plugin erstellt. Im nächsten Schritt wird diese Annotation, die auf Grund der Vererbungsbeziehung zu `Annotation` aus dem Komponentenmodell gültig ist, in eine Instanz des Komponentenmodells mittels des *Shortcut*-Mechanismus importiert. Daraufhin ist eine beliebige Entität des Komponentenmodels mit der neu geschaffenen Annotation assoziierbar.

Die Funktionsweise ist für SEFFs und Protokolle in gleicher Weise möglich – unter Umständen wird dabei, so wie in den Beispielmodellierungen von FSMs und Petrinetzen, eine Annotation importiert, die als Wrapper den tatsächlichen Annotations-Typ kapselt. Um den *Shortcut*-Mechanismus auf externe Annotationen, SEFFs und Protokolle anwenden zu können, müssen diese sich für die Serialisierung an die folgenden Datei-Erweiterungen halten, um vom GMF-Editor (Diagramm-*Plugin*) für das Komponentenmodell erkannt zu werden:

- `PcmSeff` für SEFFs
- `PcmProtocol` für Protokolle
- `PcmAnnotation` für Annotationen

Diese Datei-Erweiterungen werden durch den GMF-Editor des Komponentenmodells festgelegt.

Synchronisation von Sichten Werden Diagramm-*Plugins* verwendet, um zur gleichen Zeit verschiedene Sichten auf *ein* Domänenmodell zu haben, werden bis jetzt durch GMF keine Sychronisationen zwischen den Sichten durchgeführt. So kann eine Entität in einer Sicht gelöscht werden – was sich auch auf das Domänenmodell auswirkt – in einer anderen Sicht wird diese Änderung jedoch erst nach einem erneuten Öffnen der Sicht dargestellt.

Modifikationen am Domänenmodell und Mappings Werden ECORE-Entitäten des Domänenmodels entfernt, so wirkt sich dies im GMF Mapping (gmfmap) lediglich auf die entsprechenden Entitäten aus. Die Referenzen zeigen daraufhin auf nicht mehr existierende ECORE-Entitäten. Werden neue Entitäten im Domänenmodell hinzugefügt, hat dies auf das GMF Mapping zunächst keine Auswirkungen.

Für den Quellcode des Diagramm-Plugins existieren gleichwohl Auswirkungen durch das Hinzufügen oder Entfernen von Entitäten des Domänenmodells. Im Regelfall ist der Quellcode nach einem neuen Erzeugungsvorgang nicht mehr kompilierbar. Dies liegt an den Eigenschaften der verwendeten *JET-Template-Engine*. Da bestehende Quellcode-Blöcke, die keine Entsprechung in einem neuen Generierungsvorgang haben, ignoriert werden, führt dies zu veralteten Quellcode-Blöcken, die in der Folge ein fehlerfreies Kompilieren verhindern.

Quellcode bleibt hingegen zumeist kompilierbar, wenn ein bestehendes Modell beibehalten wird und lediglich Transformationsanweisungen im GMF Mapping modifiziert werden – ohne hier neue Entitäten hinzuzufügen oder zu löschen. Auch manuelle Modifikationen des Quellcodes bleiben in diesen Fällen zuverlässig erhalten, sofern die entsprechenden Stellen mittels „generated not“ gekennzeichnet sind.

Auswahl von Attributen In den *Wizards* von GMF und dem EMF-Editor werden jegliche Attribute von ECORE-Klassen mit ihrem nicht qualifizierten Namen angegeben. Haben etwa zwei Entitäten das Attribut „description“, so ist nicht zuzuordnen, von welcher Entität dieses Attribut stammt. Da GMF grundsätzlich Drop-Down-Menüs verwendet, besteht keine Möglichkeit, die gleichnamigen Attribute zu unterscheiden.

Aus diesem Grund wurde das Komponentenmodell angepasst. Jegliche Attribute von Klassen wurden mit einem eindeutigen Namen versehen. Zusätzlich wurden die Endpunkte aller Assoziationen, Aggregation und Kompositionen (die ebenfalls in Klassenattribute transformiert werden) umbenannt, um eine eindeutige Zuordnung zu ermöglichen. Als Muster wurde `[Alter-Attribut-Name]__[Entität-zu-der-das-Attribut-zugeordnet-ist]` verwendet.

Generierung über Wizards Die Verwendung der vollständig von *Wizards* gestützten Generierung eines GMF-Editors für das Komponentenmodell funktioniert nicht. Ein Generierungslauf für den GMF-Editor, der alle Attribute von Entitäten mit dem *Wizard* erzeugte, führte stets zu nicht kompilierbaren Quellcode. Offenbar kommen die *Wizards* von GMF (noch) nicht mit der Komplexität des Komponentenmodells zurecht. Wurden die *Wizards* zur Erstellung von gmfgraph und gmftool angewiesen, lediglich für Entitäten Transformationsanweisungen zu generieren, nicht jedoch für Attribute, blieb der Quellcode fehlerfrei generierbar. Auch eine Anwendung des *Wizards* für gmfmap für *alle* Entitäten und Attribute führte zu reproduzierbaren Fehlern.

Wurden die auf diese Weise von den *Wizards* nicht generierten fehlenden Attribute später manuell in den Generierungsanweisungen von GMF ergänzt, konnte der Quellcode fehlerfrei generierbar gehalten werden. Da aus den bereits oben genannten Einschränkungen bei der Zuweisung von *Tools* ohnehin größere manuelle Anpassungen notwendig waren, empfahlen sich die *Wizards* im Rahmen der Diplomarbeit nur zur Erzeugung einer Grundstruktur für den GMF-Editor.

Übersicht der Wizards Die von GMF bereitgestellten *Wizards* verlieren an Übersichtlichkeit für größere Modelle wie das Palladio Komponentenmodell. Da stets alle Attribute und Entitäten eines Modells in einem Editor-Schritt bearbeitet werden müssen, ist die Eignung für große Modelle nicht gegeben.

Duplizierte Attribute über Wizards Der *Wizard* für gmfgraph nimmt implizit an, dass jegliche Attribute, auch solche, die ererbt wurden, einer eigenen Darstellung bedürfen. Das führt dazu, dass für die Attribute `id` und `entityName`, die von allen Entitäten ererbt werden, jeweils eigene Label angelegt werden. Der Name der erzeugten Label richtet sich jedoch stets nach dem dargestellten Attributnamen, so dass dies zu einer Definition von gmfgraph führt, in der für jede Entität dupliziert nicht unterscheidbare Label existieren. Mit der bereits angesprochenen Problematik der Re-

ferenzierung in ECORE über den Namen, ist damit nicht entscheidbar, welches Label tatsächlich verwendet wird.

Nach der Erzeugung von gmfgraph sollte daher darauf geachtet werden, dass `IDLabel` und `EntityNameLabel` jeweils genau einmal in gmfgraph definiert werden.

Link-Tools *Tools* die in GMF *Links* erzeugen, funktionieren stets nur in eine Richtung, sofern die Klassen, die assoziiert werden, nicht vom gleichen Typ sind. So kann eine Rolle (*Link*) – je nach Definition – nur von einem *Interface* zu einer Komponente gezogen werden, nicht jedoch umgekehrt. Hierfür wäre ein weiteres *Tool* notwendig, bei dem Ziel und Quelle des *Links* vertauscht wären. Insbesondere bei ungerichteten Assoziationen wäre ein Mechanismus wünschenswert, der es erlaubt, zwei Richtungen zur Erzeugung von Assoziationen auf das gleiche *Tool* abzubilden.

Integrierter EMF-Editor Zu allen Entitäten des Domänenmodells bietet GMF eine Attribut-Sicht im *Property Sheet*, die ähnlich der des generierten EMF-Editors ist. Für Attribute die graphisch editierbar sind, wird die Bearbeitungsfunktion jedoch entfernt. Insbesondere bei größeren Modell-Instanzen, wenn eine graphische Notation unübersichtlich wird, besteht damit keine Möglichkeit, Attribute nicht-graphisch einzusehen oder zu manipulieren.

Diverses

- Toolbar-Erweiterungen, wie „Separator“, die bereits als Elemente von gmftool vorgesehen sind, werden von GMF nicht beachtet[2].
- Trotz der Ausgabe von „Code gen successful“ beim Generierungsvorgang für Code können noch Fehler im Code existieren – oder der Code wurde nicht vollständig erzeugt.
- Das `ListLayout`-Attribut in gmfgen-Dateien wird nach neuem Generieren von gmfgen-Dateien überschrieben.

Notwendige Änderungen am Komponentenmodell Um die Funktionsfähigkeit des Komponentenmodells mit GMF sicherzustellen, mussten Änderungen am ECORE- bzw. UML2-Modell vorgenommen werden – wie zum Teil bereits diskutiert wurde.

- `Context`, `ContextComponent`, `ContextProvidedRole` und `ContextRequiredRole` erben von `Entity` und sind damit direkt von der `Factory` erzeugbar. Diese Änderung war notwendig um eine Kontext-*Figure* direkt auf der Zeichenfläche erzeugen zu können.

  Zusätzlich führen nicht mehr Kontext-Komponenten den Kontext als Komposition, sondern der Kontext komponiert sich aus Kontext-Komponenten.
- Annotation sind direkt von der *Factory* erzeugbar.
- Die Bezeichner aller Assoziationen und Attribute wurde modifiziert um im gesamten Komponentenmodell eindeutig zu sein.

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=139395

5.4. Fazit – GMF

5.4.1. Aufwand zur Nachpflege bei Modelländerungen

Quellcode Die Idee des *Model Driven Developments* steht und fällt mit dem Aufwand, der zur Erzeugung neuer Programmversionen notwendig ist. Eine Abschätzung des Aufwands für das Komponentenmodell nach Änderungen am UML2-Modell mit den Auswirkungen auf das EMF-Modell wurde bereits gegeben. Da GMF zur Erzeugung von Quellcode – wie auch EMF – die *JET Template Engine* verwendet, führen auch hier umfangreiche Änderungen am Modell zur Notwendigkeit, den Quellcode zunächst zu löschen und dann komplett neu generieren zu lassen. Der Aufwand, um fehlerhaft generierten Quellcode in Folge von Modelländerungen manuell zu überarbeiten, übersteigt in den meisten Fällen den Aufwand gegenüber einem neuen Generatorlauf mit anschließender Nachpflege manueller Änderungen – so zeigte es die Erfahrung mit dem Komponentenmodell.

Unabdingbar ist dabei eine präzise Dokumentation der Codestellen, die gegenüber einer generierten Version modifiziert wurden. Umfangreichere Modifikationen des generierten Quellcodes sind – auch aus Gründen einer klaren Modularisierung – in eigene Pakete auszulagern, womit sie bei einer Löschung des generierten Quellcodes ausgelassen werden können.

Mit steigender Zahl der manuellen Änderungen am generierten Quellcode steigt der Aufwand für einen neuen Generierungslauf zur Erzeugung einer neuen Editor-Version nach einer Modelländerung. Alternativ dazu lassen sich auch die von GMF verwendeten *JET-Templates* modifizieren. Änderungen, die in den Templates vorgenommen werden, bleiben dauerhaft – auch nach der Löschung des gesamten generierten Quellcodes – erhalten. Da die manuellen Anpassungen für den im Rahmen der Diplomarbeit erzeugten Editor lediglich einen begrenzten Umfang hatten, wurde diese Alternative nicht näher untersucht.

Manuelle Änderungen am generierten Quellcode sind nur dann notwendig, wenn ein GMF-Editor in einer Weise verändert werden soll, die von GMF nicht vorgesehen ist. Wie in diesem Kapitel dargestellt wurde, sind nahezu alle für einen GMF-Editor des Komponentenmodells benötigten Funktionalitäten direkt über GMF realisierbar, so dass grundsätzlich keine manuellen Modifikationen des Quellcode notwendig sind. Einzig die Einführung eines Signatur-Editors durch Ersetzung des *Property-Sheets* von EMF würde vermutlich eine manuelle Nachbearbeitung des generierten Quellcodes notwendig machen.

Transformationsanweisungen Mit der Verwendung einer neuen Version eines Domänenmodells in einem bestehenden GMF-Projekt können auch Einschränkungen in der Verwendung bestehender Transformationsanweisungen (gmfgraph, gmftool, gmfmap, gmfgen) auftreten. Zwar arbeiten die Transformationen von GMF – wie beschrieben – in der Regel auf den Namen der Entitäten des Domänenmodells, werden jedoch umfangreichere Änderungen am Domänenmodell vorgenommen, kann auch diese Form der Referenzierung fehlschlagen. Der Änderungsaufwand nach Änderungen am Domänenmodell richtet sich etwa linear nach der Zahl umbenannter Entitäten und Attribute. Werden Entitäten oder Attribute umbenannt, schlägt die Referenzierung über den Namen für diese Entitäten und Attribute fehl. In der gleichen Weise wirkt sich das Löschen

von Entitäten und Attributen aus. Nicht mehr vorhandene Entitäten und Attribute können nicht mehr referenziert werden und neu geschaffene Entitäten und Attribute werden nicht automatisch referenziert.

Änderungen am Domänen-Model, die sich auf Entitäten auswirken, die im Editor als *Nodes* dargestellt werden, sind am teuersten. *Nodes* haben gegenüber *Links* eine eigene *Figure* und müssen im *Mapping* gmfmap ein eigenes *Node-Mapping* sowie eventuell beabsichtigte *Compartments* definieren.

5.4.2. Bewertung von GMF

Im vorigen Abschnitt wurde bereits der Änderungsaufwand für neue Modellversionen unter Benutzung von GMF diskutiert. Die Evaluation von GMF offenbarte ein vielseitiges, änderungsfreundliches und umfassendes Framework zur Generierung von graphischen Editoren zu einem gegebenen Domänenmodell. Einige der festgestellten Einschränkungen (wie die fehlende vollständige Unterstützung für mehrere Sichten) sind als zeitweilige Einschränkungen zu sehen, die mit dem Erscheinen zukünftiger Releases behoben sein sollten. In jedem Fall besteht mit GMF die Möglichkeit, vergleichsweise schnell einen Editor neu zu generieren, wie der folgende Vergleich zeigt.

5.4.2.1. Vergleich mit Ride.NET

Der Vergleich mit den Erfahrungen aus der Projektgruppe Ride.NET [23], die ebenfalls einen graphischen Editor zu einer älteren Version des Komponentenmodells erzeugte – jedoch manuell programmiert, zeigt, dass der Aufwand zur Erzeugung eines Editors mit GMF bereits bei nur *einer* Modelliteration kleiner ist. Hierbei sind noch keine Effekte berücksichtigt, die zu erwarten sind, weil mit der Evolution des Komponentenmodells unter GMF nur ein kleinerer Aufwand für die Änderung der Definition von Transformationsanweisungen zu erwarten ist.

In Tabelle 5.1 erfolgt ein Vergleich des Ride.NET-Editors (ohne Berücksichtigung von Im- und Export-*Plugins*) mit einem mittels GMF erzeugten Editor. Ein exakter Vergleich des Entwicklungsaufwands zwischen Ride.NET und GMF erscheint unangemessen, da Ride.NET in einer Projektgruppe mit elf Personen über den Zeitraum von einem Jahr durchgeführt wurde. Neben dem graphischen Editor wurde zahlreiche *Plugins* zum Im- und Export entwickelt, die eine genaue Aufschlüsselung des jeweiligen Aufwands sehr ungenau machen würden. Zudem setzte Ride.NET auf einem prototypischen Editor auf, während im Rahmen dieser Diplomarbeit nur bis zu einem prototypischen Stadium entwickelt wurde.

Wie aus der Tabelle ersichtlich ist, werden neben Faktoren wie dem Aufwand auch die Qualitätseigenschaften von Software im Allgemeinen verbessert. Der von Generatoren erstellte Quellcode kann gegenüber Eigenentwicklungen genauer getestet werden und ist dadurch in der Regel weniger fehlerhaft. Durch eine Vielzahl von Anspruchsgruppen, die Quellcode aus den gleichen Templates verwenden wollen, werden zudem Erweiterbarkeit, Wartbarkeit und eine umfassende API gefördert.

Wichtige Unterschiede im Vergleich von Ride.NET mit dem mittels GMF erzeugten Editor erwachsen aus der Verwendung von Frameworks. Eclipse stellt Mechanismen zum Laden von Plugins und Undo / Redo Funktionalität bereit. Über EMF wird die Erzeugung einer Implementierung des Domänenmodells sowie Klassen zum Bearbeiten des Modells übernommen. Der durch GMF generierte Quellcode für GEF ermöglicht

Kriterium	Ride.NET	GMF-Editor
Programmiersprache	C#	Java
Implementierung	händisch	Generierung
GUI-Bibliothek	Netron („buggy“)	GEF („ausgereift“)
Basisframework	.NET / Netron	Eclipse / EMF / GMF / GEF
Undo / Redo	händisch erzeugt	durch Frameworks automatisch vorgesehen
Event-Erzeugung	händisch erzeugt	durch Frameworks automatisch vorgesehen
Erweiterbarkeit durch Plugins	vorhanden	vorhanden
Entwicklungsaufwand	hoch	je nach dem Grad der individuellen Anpassungen mittel bis hoch
Aufwand bei Modelländerung	hoch	mäßig (abhängig vom Grad individueller Anpassungen)
API des verwendeten Komponentenmodells	bedürfnisgerecht	umfassend generiert
Fehlerwahrscheinlichkeit	hoch	gering (Abhängigkeit vom verwendeten Release GMFs)
Unit Test	vollständig manuell zu definieren	Generierung von Teststümpfen möglich
Anpassung an individuelle Anforderungen	vollständig möglich	möglich, soweit von GMF vorgesehen; sonstige Anpassungen: manuell
Lernaufwand	hoch – Frameworks müssen komplett erlernt werden	mäßig – MDA-Werkzeuge abstrahieren Komplexität der Frameworks

Tabelle 5.1.: Vergleich von Ride.NET mit einem über GMF erzeugten Editor für das Komponentenmodell

schließlich den Zugriff auf ein ausgereiftes GUI-Framework, das bereits erfolgreich in kommerziellen Editoren verwendet wird. Die Entwicklungen von Ride.NET erfolgten im Vergleich dazu nahezu vollständig händisch. Die Wahl der .NET-Plattform mit C# für Ride.NET ließ keine Rückgriffe auf umfassende Frameworks zu, da freie Alternativen zu EMF, GMF und GEF nicht verfügbar waren.

5.4.2.2. Eignung von GMF

Auch wenn GMF nicht an ein konkretes Domänenmodell gebunden ist, ist die Nutzung dennoch auf eine spezielle Domäne festgelegt. GMF bietet einen rein graphischen Editor, der zwingend einem ECORE-Domänenmodell folgen muss. Textbasierte oder ähnliche Editoren sind von GMF nicht vorgesehen. Die Art und Weise der Darstellung der Elemente eines Domänenmodells mit Knoten und Assoziationen zwischen Knoten impliziert vor allem im Bereich komplexer Datentypen Einschränkungen. Die Darstellung komplexer Datentypen wie einer Verteilungsfunktion ist von GMF selbst nicht vorgesehen. Dazu bedarf es eines Eingriffs in den Quellcode, mit dem externe Editoren für komplexe Datentypen verfügbar gemacht werden. Im Falle des Komponentenmodells wurde dies bereits mit einem Editor für Schnittstellen-Signaturen angedeutet.

Ebenso eignet sich GMF zunächst nicht zur Umsetzung von Roundtrip-Funktionalität zu *Instanzen* eines Domänenmodell. Der Rückgriff auf EMF zur Realisierung der Implementierung des Domänenmodells bindet die Serialisierung an die von EMF intendierte XML-Form. Die Erzeugung von Quellcode zu Instanzen eines Domänenmodells oder der Import aus bestehendem Quellcode ist weder in EMF noch in GMF vorgesehen.

Die Stärken von GMF liegen somit vor allem im Bereich rein graphischer Editoren, um die Beziehungen zwischen Instanzen der Entitäten eines Domänenmodells zu modellieren. Durch die Möglichkeit, (vergleichsweise) schnell einen Editor zu einem beliebigen Domänenmodell erstellen zu können, sofern dieser keiner besonderen individuellen Anpassungen bedarf, ergibt sich die Chance, bereits während der Erstellung eines Domänenmodells (also während der Meta-Modellierung) einen intuitiven Zugang zu einer Domäne über die Arbeit mit einem GMF-Editor-Prototypen herzustellen. Sind häufig Instanzen sich wandelnder, entwickelnder oder vollständig neuer Domänenmodelle zu modellieren, kommen die MDA-Wurzeln von GMF besonders zu Tragen.

Über die Möglichkeiten GMF-Editoren – gleichwohl unter Mehraufwand – an individuelle Bedürfnisse anzupassen, zeigt sich die Eignung von GMF auch für angepasste Editoren. Hierbei ist zu beachten, dass häufige Änderungen am Domänenmodell auch unter GMF zu mehr Aufwand bei der Entwicklung neuer Editor-Versionen führen. Je kleiner der Grad von Anpassungen der Standard-Werte für die Transformationen ausfällt, desto einfacher lassen sich neue Editor-Versionen erstellen. Wie der Vergleich mit Ride.NET zeigt, fällt die Aufwand zur Erstellung neuer Versionen unter GMF dennoch geringer aus als bei konventioneller Entwicklung.

Neue Releases von GMF lassen überdies weniger Fehler und Einschränkungen durch Fehler, wie sie in Kapitel 5.3.4.4 genannt wurden, erwarten. Neue Features sind indes erst für Versionen >1.0 zu erwarten.

6. Fazit

6.1. Zeitplanung

Vergleich man die Zeitplanung, wie sie ursprünglich im Proposal der Diplomarbeit [44] vorgestellt wurde, in der Retrospektive mit der realen Umsetzung, so sind zahlreiche Unterschiede festzustellen.

Die in den Überschriften zu den Phasen angegebenen Zeitpunkte geben grob den zeitlichen Rahmen der Phasen an. Die Phasen lassen sich nicht klar voneinander trennen, da immer wieder Anpassung in den Themenbereichen vorangehender Phasen – entsprechend einer iterativen Entwicklung – notwendig waren.

Erste Phase – Dezember 2005 In der ersten Phase der Diplomarbeit fand eine Auseinandersetzung mit den technischen Möglichkeiten zur Realisierung der angestrebten Modellierung und anschließenden Transformationen statt. Es erfolgte die Evaluation verschiedener Modellierungswerkzeuge (etwa Together Architect, RSA und Omondo UML) sowie verschiedener Frameworks für die Transformation des Komponentenmodells zu einem graphischen Editor. Hier wurden die auf Eclipse basierenden Projekte EMF, GMT, GEF, UMLX und Merlin auf ihre Eignung für die Diplomarbeit hin getestet. Zusätzlich wurden die Transformationsmöglichkeiten, mit denen sich aus dem schließlich gewählten RSA UML2-Diagramme exportieren und in EMF importieren ließen, untersucht. Zusätzlich wurden die zum damaligen Zeitpunkt verfügbaren Erzeugungsmöglichkeiten von GMF für einen grafischen Editor für einfache Modelle erprobt.

Aus dieser Phase ging eine Entscheidung für die zu verwendenden Tools und Frameworks hervor. Zudem konnte über die prototypische Umsetzung des in der Diplomarbeit angestrebten Entwicklungsprozesses eine Minimierung des Risikos für die Diplomarbeit erwirkt werden. Da der gesamte Prozess – wenn auch mit einfacheren Modellen – bereits im Vorfeld durchgeführt wurde, konnte die Entwicklung des UML2-Modells in den späteren Phasen sehr gezielt erfolgen.

Zweite Phase – Januar 2006 Entgegen der ursprünglichen Planung wurde in der zweiten Phase noch keine iterative UML2-Modellierung mit anschließender Transformation mittels EMF zu einem Java-Modell gestartet, sondern eine ausführliche Anforderungsermittlung durchgeführt. Das Palladio Komponenten-Meta-Modell existierte zum Start der Diplomarbeit in keiner aktuellen und vollständigen Version in schriftlicher Form. Daher waren weder Annahmen, Konstrukte, Konzepte noch Constraints vollständig schriftlich verfügbar. Insbesondere das neu eingeführte Konzept des Kontexts war bis zu diesem Zeitpunkt wenig klar formuliert. Aus diesem Grund wurde nach Klärung der technischen Voraussetzungen zunächst eine genaue Ermittlung der Ideen des Komponentenmodells betrieben. Dafür wurde die in Kapitel 2 zu findende Beschreibung des Komponentenmodells in mehreren Iterationen verfasst. In zahlreichen Rück-

kopplungen wurden die genauen Konzepte und Constraints des Komponentenmodells mit den Mitgliedern der Palladio-Gruppe diskutiert und anschließend festgehalten.

Im Ergebnis lieferte diese Phase das Kapitel 2. Dieses hält ausführlich das Palladio Komponentenmodell mit samt den derzeit bekannten Einschränkungen und Variationspunkten fest. Zusätzliche erklärende Anmerkungen sollten das Kapitel ebenfalls als zukünftige Diskussionsbasis und Ausgangspunkt für die Vermittlung des Komponentenmodells an zukünftige Studierende geeignet machen.

Dritte Phase – Februar 2006 In der dritten Phase der Diplomarbeit wurde das Komponentenmodell, entsprechend der Beschreibung der vorigen Phase, in ein UML2-Modell unter RSA umgesetzt. Diese Phase entsprach wieder der ursprünglichen Planung aus dem Proposal der Diplomarbeit. In mehreren Iterationen wurde das UML2-Modell um zusätzliche Konzepte erweitert. Damit der Import in EMF und die dortige Transformation zu einem Java-Modell und einem Editor funktionierte, wurde die Generierung von Java-Modell und Editor stets getestet.

Zuletzt wurden die Constraints für das Komponentenmodell mittels OCL spezifiziert. Entgegen der ursprünglichen Annahmen konnte jedoch keine Validierung der OCL-Constraints an Hand von Instanzen des Komponentenmodells erfolgen. Ausschlaggebend für diese Einschränkung war der fehlerhafte und unvollständige Export mehrerer Sub-Modelle einschließlich ihrer Constraints aus RSA heraus, sowie die gänzlich fehlende Unterstützung von OCL in den *Stable*-Entwickler-Versionen von EMF für die zum Zeitpunkt der Durchführung verfügbaren Versionen.

Als Resultat dieser Phase sind folgende Artefakte festzustellen:

- RSA-UML2-Modell-Projekt inklusive spezifizierter OCL-Constraints
- Aus RSA heraus exportierte Versionen des UML2-Modells – jedoch exklusive OCL-Constraints
- Dokumentation des UML2-Modells sowie der OCL-Constraints

Diese Phase konnte zeitlich entsprechend der Planungen im Proposal abgeschlossen werden.

Vierte Phase – Februar 2006 Diese Phase widmete sich der Transformation der aus RSA exportierten UML2-Sub-Modelle (PalladioCM, IdentifierModel, FsmWrapper, usw.) mittels EMF. Durch die stetigen Tests in der vorangehenden Phase konnte diese Phase schnell durchgeführt werden. Nach Ergänzung der exportierten Modelle um Identifier mittels UUID sowie der vorangehenden Erprobung der Alternativen zur Modellierung von Identifiern (wie oben beschrieben), wurde diese Phase mit der Dokumentation abgeschlossen. Dies beinhaltet ebenfalls den von EMF generierten Editor zur Bearbeitung von Instanzen des Komponentenmodells.

Fünfte Phase – März/April 2006 In dieser fünften Phase der Diplomarbeit wurde die Erstellung eines graphischen Editors mittels GMF durchgeführt. Die Einarbeitung in das Framework sowie die Eingrenzung von Fähigkeiten und Fehlern des Frameworks war zeitintensiv. Der Wechsel zwischen verschiedenen *Milestone-Releases* (M4 → M5 → M6) machte auf Grund geänderter Datenserialisierung mehrfach eine komplette

Neu-Definition der Transformationsanweisungen notwendig. Auch die Modifikation des Komponenten-Meta-Modells zur Anpassung an GMF führte zur Notwendigkeit, die Transformationsanweisungen neu zu definieren. Auf Grund massiver Fortschritte, war die Arbeit mit späteren *Releases* schneller durchführbar – diese enthielten erwartungsgemäß deutlich weniger Fehler als frühe Versionen. Die Ergründung einiger Fehler, etwa bei der Aufteilung auf verschiedene Sichten auf das Komponentenmodell, machte intensive Diskussionen mit den Entwicklern von GMF notwendig.

Insgesamt machte diese Phase, trotz des Vorhabens Konzepte von GMF lediglich prototypisch zu ergründen, auf Grund der Vielzahl zu untersuchender Konzepte einen größeren Teil der Arbeit aus, als zunächst angenommen. Für eine mögliche komplette Umsetzung eines angepassten Editors unter GMF wäre vermutlich ein Zeitbedarf von mehr als vier Monaten zu veranschlagen.

Sechste Phase – April 2006 Durch die kontinuierliche Dokumentation parallel zu den vorangehenden Phasen, waren hier vor allem einleitende und abschließende Abschnitte zu erstellen. Zusätzlich war Zeit für die Durchführung von Korrekturen und das Drucken und Binden zu veranschlagen. Durch die Parallelisierung von Entwicklung und Dokumentation fiel der zeitliche Bedarf dieser Phase letztlich kleiner als angenommen aus.

6.2. Bewertung des MDA-Ansatzes

Das Ziel der vorliegenden Diplomarbeit bestand neben der Modellierung des Palladio Komponenten-Meta-Modells in der Anwendung von Transformationen auf das erstellte Meta-Modell. Insgesamt wurden zwei grundlegende Transformationsschritte des Modells durchgeführt. Der erste Schritt umfasste die Transformation des in UML2-Modell repräsentierten Meta-Modells über die Verwendung von EMF in ECORE sowie dazugehörigem Java-Quellcode. Im zweiten Schritt wurde das auf diese Weise geschaffene ECORE-Modell inklusive des dazugehörigen Java-Quellcodes verwendet, um mit GMF einen graphischen Editor für das Meta-Modell zu erzeugen.

Auf diese Weise wurde ein kompletter MDA-Prozess durchlaufen, dessen Transformations-Übergänge so weit wie möglich automatisiert und reproduzierbar angelegt waren.

6.2.1. Informationsergänzungen

Grundsätzlich gilt, dass Transformationen einem Modell Informationen hinzufügen können. Alle für das Komponentenmodell durchgeführten Transformationsschritte waren so angelegt, dass sie Informationen ergänzten, die letztlich zur Erzeugung eines graphischen Editors notwendig waren. Bei der Transformation von UML2 zu ECORE wurden die in ECORE zusätzlich vorhandenen Attribute mit Standard-Werten belegt. Bei der Erzeugung einer Repräsentationen des Komponentenmodells in Java-Quellcode wurden Erzeugungsroutinen, Validierungsmöglichkeiten und beispielsweise Event-Behandlung ergänzt. Über die mittels GMF vorgenommenen Transformationen wurden unter anderem graphische Repräsentationen mit GEF, *Tools* und *Plugin-Wizards* als Informationen ergänzt.

Können die Anforderungen an das Ergebnis eines MDA-Projekts über die von Transformationswerkzeugen vorgesehenen Informationsergänzungen erreicht werden, ist folg-

lich der Aufwandsgewinn von MDA gegenüber einer manuellen Implementierung des gleichen Projekts besonders groß.

6.2.2. Transformationsschritte

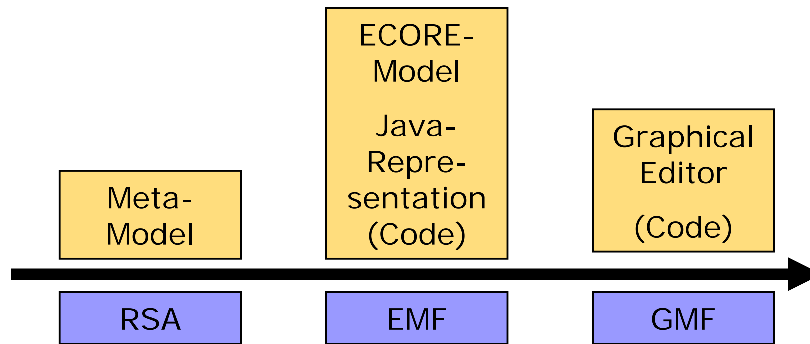


Abbildung 6.1.: Zuordnung von Transformationsschritten zu Werkzeugen

Abbildung 6.1 beschreibt die drei verwendeten Haupt-Technologien für die durchgeführten Transformationsschritte sowie die vorliegenden Formen von Artefakten. Über RSA wurde zunächst ein reines Meta-Modell in UML2 erstellt. Über einen Export aus RSA und Import in EMF wurde das Meta-Modell in ein ECORE-Modell transformiert (Model-2-Model-Transformation). Zusätzlich wurde innerhalb von EMF aus dem ECORE-Modell über eine Model-2-Text-Transformation eine Java-Repräsentation erzeugt. Für die Transformationen durch GMF dienten ECORE-Modell und Java-Repräsentation gleichermaßen als Eingabe. In GMF sind Model-2-Model und Model-2-Text Transformationen kombiniert. Da GMF entsprechend des eingegebenen ECORE-Modells eigene interne Modelle zum Hinterlegen der Transformationsanweisungen generiert, handelt es sich hierbei um eine Model-2-Model-Transformation. Die Funktionen GMFs zum Generieren des Diagramm-Editor-Codes entspricht einer Model-2-Text-Transformation.

Der Grad, zu dem die oben genannten Transformationen automatisiert durchgeführt werden konnten, variiert jedoch stark. Insbesondere die Menge notwendiger Transformationsanweisungen zur Erreichung der gewünschten Resultate unterschied sich deutlich:

- Die Transformation von UML2 aus RSA heraus zu ECORE benötigte kaum ergänzende Anweisungen. Das erzeugte ECORE-Modell hatte lediglich Defizite im Bezug auf *Identifier*. Da UML2 für Klassendiagramme kein Klassen-Attribut unterstützt, das Informationen zum in der Serialisierung zu verwendeten Referenzierungsmechanismus für Instanzen („Identifier“) untereinander definiert, musste diese Information händisch ergänzt werden. Die Transformation von UML2 zu ECORE und der Import in EMF erfolgte voll automatisch und erlaubte daher nicht das Hinterlegen von Transformationsanweisungen, durch die eine händische Anpassung nicht notwendig gewesen wäre. Würden Transformationsanweisungen (außerhalb der *JET-Templates*) definierbar sein, könnte die genaue Belegung von ECORE-Attributen für jede Meta-Klasse über die Transformationsanweisungen geschehen. Eine manuelle Anpassung wäre demnach nicht notwendig.

Das Hauptproblem besteht – neben den fehlenden Anpassungsmöglichkeiten des Transformationsvorgangs – in den unterschiedlichen Attributen, die UML2- und ECORE-Klassen definieren können. Dies betrifft, wie oben genannt, die Behandlung von Identifiern aber auch beispielsweise die Steuerung der erlaubten Erzeugungsprozesse in Modell-Instanzen. Da die Belegung von ECORE-Attributen mit Nicht-Standardwerten in der Transformation von RSA nicht vorgesehen ist, sind nachträgliche manuelle Änderungen notwendig.

Die für das Komponentenmodell notwendige manuelle Änderung des Attributs für Identifier bedeutete einen geringen Aufwand im Vergleich zum oben skizzierten Gesamtprozess.

- Bei der Erzeugung von Java-Quellcode unter EMF als Repräsentation des Komponentenmodells waren für die Model-2-Text-Transformation Eingriffe möglich. Als Einschränkung für den generierten Java-Quellcode ergab sich für das Komponentenmodell, wie in Kapitel 4.3 beschrieben, jedoch dennoch die Notwendigkeit den Quellcode zur Initialisierung eines eindeutigen Identifiers manuell zu modifizieren, da die notwendigen Anpassungen für die Transformation nicht vorgesehen waren.

  Auf Grund der Entkopplung zwischen dem Identifier-Meta-Modell und dem restlichem Komponentenmodell waren, wie in der Diplomarbeit beschrieben, keine fortwährenden manuellen Nachbearbeitungen des Quellcodes notwendig, so dass der Gesamtaufwand für die Anpassung des Quellcodes als gering anzusehen ist.

- Die Transformationen in GMF teilten sich, wie bereits im vorangegangenen Kapitel ausführlich erläutert wurde, auf mehrere Schritte auf. In den entsprechenden Transformationsanweisungen ließen sich außer den zur Transformation des Komponenten-Meta-Modells benötigten Informationen zahlreiche ergänzende Anweisungen zur Generierung des Diagramm-Plugins hinterlegen. Große Teile der Transformationsanweisungen waren nach der Ausführung der GMF-*Wizards* nicht mit sinnvollen Werten belegt. Zudem erforderte der intendierte graphische Editor weitere Modifikationen der bestehenden Standard-Einstellungen.

  Insgesamt waren für die unter GMF erfolgten Transformationsschritte die umfangreichsten Transformationsanweisungen notwendig. Grundsätzlich mussten Anweisungen zu jeder Klasse aus dem ECORE-Komponentenmodell (Graphik, Tool, Mapping, Generierung) angelegt werden. Zusätzlich war die Definition von Konstrukten wie *Compartments* und die Erzeugung von *Shortcuts* notwendig, um den gewünschten Editor generieren zu können. Grob abgeschätzt gab es einen Basis-Aufwand für die Verwendung von GMF, sowie einen etwa linear von der Anzahl der Modellelemente abhängigen Zusatzaufwand.

  Neben dem Aufwand zur Erstellung der Transformationsanweisungen ist zu beachten, dass auch unter GMF Änderungen am generierten Java-Quellcode (in diesem Falle des Diagramm-*Plugins*) notwendig waren. Der abzuschätzende Aufwand richtet sich vor allem nach der Anzahl individuell, das heißt nicht mit den von GMF vorgesehenen Mitteln, bearbeitbarer Attribute und Assoziationen des Komponentenmodells und der Anzahl graphisch individuell dargestellter Entitäten des Komponentenmodells.

Unter den genannten Gesichtspunkten und als Folgerung aus dem Fallbeispiel des Komponentenmodells erscheint die Anwendung von MDA vor allem für jene Anwendungsbereiche interessant, die eine geringe Anpassung der von den verwendeten Werkzeugen vorgesehenen Transformationsschritte benötigen. Zusätzlich sollten nur wenige manuelle Modifikationen von Quellcode zur Erreichung des gewünschten Zieles notwendig sein. Übersteigt der Bedarf an Änderungen die Flexibilität der verwendeten Transformationswerkzeuge, bedarf es stets manueller Anpassungen des Transformationsvorgangs oder der Resultate des Transformationsvorgangs, die die Vorteile eines MDA-Ansatzes verringern.

Wie bereits im Kapitel zu GMF im Vergleich zwischen generiertem Editor und Ride.NET-Editor aufgezeigt wurde, erscheint die Wahl eines MDA-Ansatzes zur Erzeugung eines graphischen Editors für das Komponentenmodell insbesondere sinnvoll, weil der Gesamtaufwand zur Erzeugung eines neuen oder weiter evolvierten graphischen Editors deutlich geringer ist, als bei einer manuellen Erzeugung.

Die Tatsache, dass es keine elementaren Konzepte des Komponentenmodells gibt, die in den verschiedenen Meta-Modell-Repräsentationen (UML2, ECORE, Java-Quellcode) und über die Transformationsschritte hinweg überhaupt nicht umgesetzt werden konnten, zeigt eine Eignung von MDA für die Domäne der Entwicklung von Softwaremodellen, wie sie am Beispiel des Komponentenmodells durchgeführt wurde.

6.2.3. Werkzeugunterstützung

Die meisten Einschränkungen bei der Durchführung der Diplomarbeit ergaben sich durch die verwendeten Werkzeuge und nicht durch die Verwendung von ECORE und UML2 zur Repräsentation des Komponenten-Meta-Modells. Wie in den Kapiteln zur Modellierung, Verwendung von EMF und GMF aufgezeigt wurde, ließen sich etwa unter RSA keine UML-Profile verwenden, wenn ein Export des Komponentenmodells erfolgen sollte. Auch die Aufteilung des Komponentenmodells auf mehrere Sub-Modelle und die Verwendung von OCL bereitete Probleme beim Export zu UML2. Die Transformationsvorgänge beim Import von EMF ließen sich nicht näher steuern. GMF brachte vor allem Einschränkungen durch seinen frühen Versionsstand.

- Gegeben ist eine schnelle und einfache Änderbarkeit der Modellrepräsentation in UML2 unter RSA.

- Durch die Auslagerung von *Identifiern* in ein eigenes Sub-Modul können *Merges* verschiedener Modellversionen unter EMF im Allgemeinen konfliktfrei abgewickelt werden.

- Modelländerungen wirken sich in der Regel unter GMF nur auf modifizierte Modellelemente aus, ohne dabei Seiteneffekte auszulösen.

- Konzeptionell werden verschiedene Sichten auf die gleiche Instanz des Komponentenmodells durch GMF bereits unterstützt.

- Eine Änderbarkeit aller Attribute des Komponentenmodells unter GMF ist – als Schlußfolgerung der prototypischen Evaluierung – möglich.

Insgesamt ist die Werkzeugunterstützung des durchgeführten MDA-Prozesses bereits jetzt als ausreichend zu bewerten, um einen Großteil der Anforderungen an einen graphischen Editor für das Komponentenmodells umzusetzen. Der Entwicklungsprozess, beginnend mit Änderungen der Meta-Modell-Repräsentation in UML2, kann sinnvoll im Bezug auf die Unterstützung der Werkzeuge durchgeführt werden.

6.3. Zielvergleich

Vergleicht man die für die Diplomarbeit gesteckten Ziele, so wie sie im Proposal [44] aufgezeigt wurden, so ließen sich diese vollständig erfüllen. Der vorgesehende MDA-Prozess (Abbildungen 3.1 und 3.2) konnte komplett durchgeführt werden.

Im Entwicklungsprozess wurde bei der Wahl zwischen Together Architect und RSA zu Gunsten von RSA entschieden und zum Generieren des Editors wurde GMF gegenüber Merlin bevorzugt. Die Entscheidung gegen Merlin und für GMF dürfte erst eine realistische Möglichkeit zum Erstellen des Editors im gewünschten Umfang ermöglicht haben, da die ansonsten in Teilen notwendige manuelle Programmierung der GEF-Komponenten des Editors den zeitlichen Rahmen der Diplomarbeit gesprengt hätte. Die Erfahrungen mit Merlin in der Auswahlphase der Werkzeuge zeigte, dass es weit weniger fortgeschrittene Transformationsmöglichkeiten gab, um einen graphischen Editor zu erzeugen.

Das Komponentenmodell wurde in seiner aktuellen Ausprägung vollständig beschrieben. Bei der Formalisierung der Beschreibung über die Modellierung des Komponentenmodells konnten konzeptionelle Einschränkungen aufgezeigt und Vorschläge für Erweiterungen eingebracht werden. Die Modellierung des Komponentenmodells mit samt der getroffenen Entwurfsentscheidungen und *Constraints* wurde ausführlich begründet.

Die Transformation des Komponentenmodells in eine ECORE-Repräsentation sowie Java-Quellcode konnte unter Darstellung der damit verbundenen Einschränkungen durchgeführt werden. Aus der Transformation resultierende Notwendigkeiten zur manuellen Anpassungen wurden dokumentiert.

Zum Abschluss des MDA-Prozesses wurde die von GMF gebotene Technologie ausführlich auf die Bedürfnisse eines graphischen Editors für das Komponentenmodell hin untersucht. Einschränkungen durch die Konzepte von GMF wurden dargelegt, Empfehlungen für den Umgang mit GMF zur Erstellung eines graphischen Editor für das Komponentenmodell wurden ausgesprochen.

Insgesamt fand eine Bewertung des MDA-Prozesses statt, wobei ein besonderes Augenmerk dem zu erwartenden Aufwand zur Durchführung des MDA-Prozesses galt. Betrachtet wurde hierbei auch die Eignung des MDA-Prozesses mit den gewählten Werkzeugen zur evolutionären Entwicklung neuer Versionen des Komponentenmodells.

6.4. Ausblick

Der Ausblick auf mögliche Weiterentwicklungen dessen, was mit dieser Diplomarbeit begonnen wurde, erstreckt sich vornehmlich auf drei Bereiche:

1. Auch zukunftig wird das Palladio Komponentenmodell weitere Änderungen erfahren. In neuen Iterationen kann daraufhin der Entwicklungsprozess, so wie er in Abbildung 3.2 skizziert wird, durchlaufen werden. Dabei werden insbesondere

die Fähigkeiten EMFs und GMFs im Umgang mit neuen Versionen des Komponentenmodells dafür von Bedeutung sein, in wie weit Änderungen mit den bestehenen Modelle „gemerged“ werden können. Entscheidend ist dafür der Umfang der Änderungen am Komponentenmodell. Insbesondere neue Versionen der in EMF und GMF verwendeten *JET-Engine* könnten im Bereich der Quellcode-*Merges* Verbesserungen bringen.

2. Der mittels GMF entwickelte graphische Editor hatte lediglich prototypischen Charakter. Eine Weiterentwicklung des bestehenden Editors, um diesen auf das Niveau eines vollständig ausgestalteten Editors zu heben, wäre denkbar. Vor allem die graphische Ausgestaltung des Editors und zahlreicher Komfortfunktionen sowie die Implementierung eines Editors für Signaturen wären als Erweiterung vorzusehen. Durch die Verwendung fortgeschrittenerer Releases von GMF könnten zudem bestehende Einschränkungen entfallen.

3. Nachdem sich diese Diplomarbeit im Kern auf die Verwendung von EMF und GMF stützte, wären weitere Arbeiten mit alternativen (neuen) Werkzeugen denkbar. Insbesondere der Vergleich verschiedener Werkzeuge untereinander könnte wichtige Hinweise auf den Entwicklungsstand von MDA-Werkzeugen im Allgemeinen liefern und damit als Indiz für den Reifegrad von MDA im Allgemeinen sein.

   Ebenfalls spannend wäre ein Vergleich der Eignung von EMF und GMF für andere Domänen als das Palladio Komponentenmodell. Zwar liegt eine Eignung für Modelle von Softwaresystemen im Allgemeinen nahe, eine Untersuchung anderer Software-Modelle und vollständig anderer Domänen steht jedoch aus. Aus der Ausweitung der Untersuchungen auf andere Domänen ließen sich weitere Rückschlüsse auf die Flexibilität von EMF und GMF ziehen.

6.5. Zusammenfassung

In dieser Diplomarbeit wurde ein Entwicklungsprozess dargelegt, der den Grundsätzen der MDA folgt.

Nachdem in der Vorbereitung bereits die Entscheidung zugunsten ausgewählter Werkzeuge gefallen war, bestand die erste Phase der Arbeit in der Konstruktion eines Meta-Modells zum Palladio Komponentenmodell. Durch zahlreiche Diskussionen in der Forschungsgruppe wurden die Konzepte des Komponentenmodells ergründet. Hieran schloss sich die Dokumentation des Komponentenmodells sowie eine parallele Modellierung in UML2 unter RSA an. Mit dem Abschluß dieser Phase lag eine erste Version des Komponentenmodells als theoretisches Konstrukt wie auch beispielhaft modelliert vor.

Das auf diese Weise erzeugte Komponentenmodell fand anschließend Eingang in die Verwendung in EMF. Hier wurde die Modellierbarkeit von Instanzen des Komponentenmodells beispielhaft getestet. Erfahrungen aus den Tests fanden wiederum Einzug in das Komponenten-Meta-Modell. An dieser Stelle sind die Umsetzung der Modellierung des Kontexts, der Identifier sowie dynamischer Aspekte mit Annotationen, SEFFs und Protokollen hervorzuheben.

Nach zufriedenstellendem Abschluss der Tests mit dem durch EMF generierten Java-Modell samt durch EMF generiertem Editor folgte die Evaluation von GMF. In GMF diente das Komponenten-Meta-Modell als Eingabe als Domänenmodell. Beginnend mit einem um viele Entitäten reduzierten Komponentenmodell wurden nach und nach die für einen Komponentenmodell-Editor gewünschten Fähigkeiten evaluiert. Zu den Kernproblemen zählten die Fähigkeiten zur Darstellung von *Compartments*, die Nutzung von Annotationen, SEFFs und Protokollen aus externen Modellen, die Aufteilung der Diagramm-*Plugins* auf verschiedene Sichten, die Untersuchung möglicher Referenzierungen zwischen verschiedenen Sichten sowie die Möglichkeiten zur Anpassung der graphischen Repräsentation. Auch in dieser Phase ergaben sich diverse Anpassungen in der Modellierung des Komponentenmodells, die aus der Verwendung von GMF resultierten. Dabei wurde stets die textuelle Dokumentation des Komponentemodells aktuell gehalten.

Diese Arbeit schloss mit einer Bewertung der Ideen und der Reife von MDA und deren Werkzeugen. Aus den Erfahrungen in der Umsetzung des MDA-Prozesses in dieser Arbeit wurden Rückschlüsse auf die Eignung von MDA für die zukünftige Entwicklung des Komponentenmodells inklusive eines dazu passenden graphischen Editors gezogen. Insbesondere wurden Abschätzungen zu Aufwand, Änderungsfreundlichkeit und Einschränkungen von MDA im Allgemeinen vorgenommen.

A. Anhang

A.1. Komponentenarten

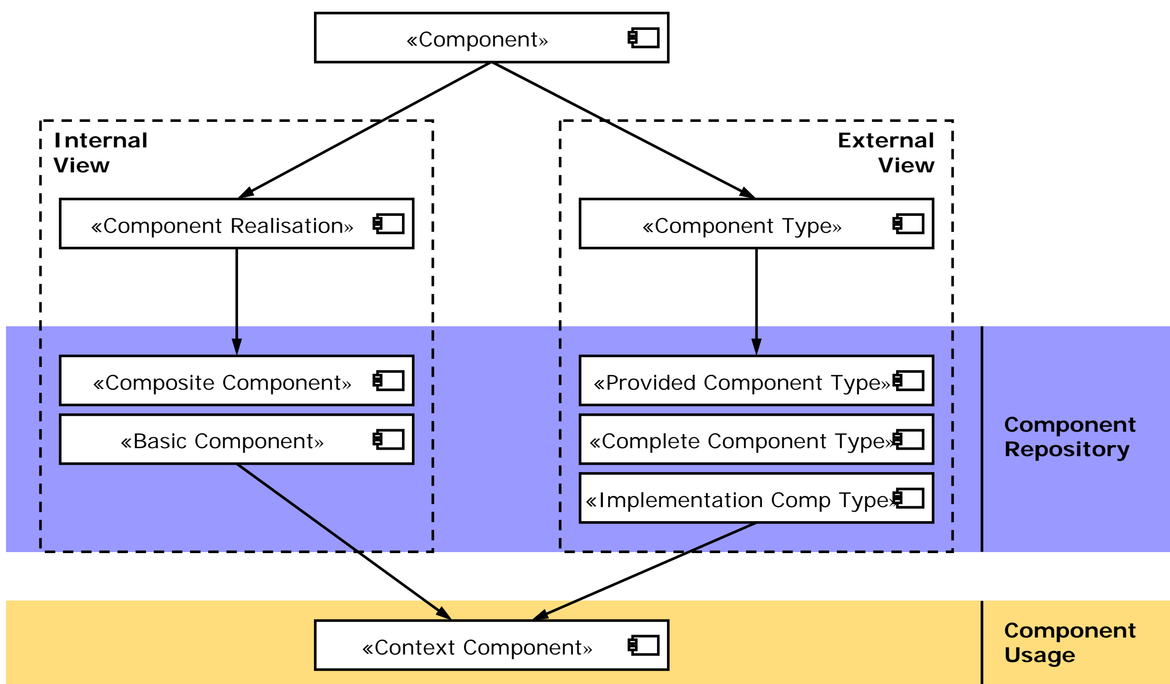


Abbildung A.1.: Vollständige Hierarchie von Komponentenarten

Abschließend soll eine zusammenfassende Übersicht über die im Komponentenmodell definierten Arten von Komponenten gegeben werden. Diese Übersicht dient insbesondere der begrifflichen Abgrenzung. Abbildung A.1 zeigt alle vorangehend beschriebenen Arten von Komponenten auf. Alle Komponentenarten sind *Components*. Diese unterteilen sich in Realisierungen (*Component Realisation*) mit nach innen gerichteter Sicht, sowie von außen betrachtet in Komponenten-Typen (*Component Type*).

Der linke Ast der Komponentenrealisierungen unterteilt sich weiter in *Basic* und *Composite Components*. Der rechte Ast führt die Komponenten der Typ-Ebenen auf: *Provided Component Type*, *Complete Component Type* und *Implementation Component Type*. Komponenten-Typen mit den beiden Formen der Realisierung als *Basic* und *Composite Component* liegen im Komponenten-Repository (*Component Repository*).

Zusätzlich ist für Komponenten-Realisierungen und Komponenten-Typen eine Verwendung von Komponenten (*Component Usage*) (in Kontexten) möglich. Dazu existiert die Kontext Komponente (*Context Component*). Diese Art von Komponenten ist jedoch weder eine Typ-Definition noch eine Realisierung, sondern lediglich eine Verwendung von Komponenten aus dem Komponenten-Repository.

A.2. Berechnung von SEFFs und Protokollen

Eine vollständige Berechnung von Protokollen und SEFFs ist nicht Gegenstand der Ausführungen in dieser Diplomarbeit. Um dennoch abgrenzen zu können, welche Berechnungen grundsätzlich möglich sind, wird in den folgenden beiden Unterkapiteln dargestellt, wie die möglichen Berechnungen durchführbar sind und welche Einschränkungen sich dadurch für manuell spezifizierte SEFFs und Protokolle ergeben.

A.2.1. Berechnung von benötigten Protokollen

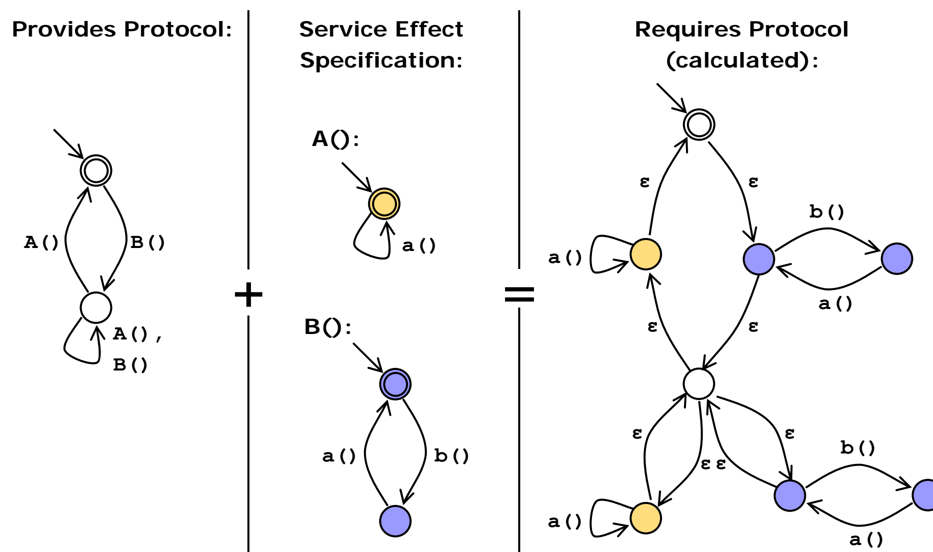


Abbildung A.2.: Beispiel der Berechnung eines *Requires Protocols* aus einem *Provides Protocol* und SEFFs

Da SEFFs eine Verbindung zwischen den Protokollen auf der angebotenen und benötigten Seite herstellen, lässt sich zu einem gegebenen angebotenem Protokoll mit Hilfe eines SEFFs ein benötigtes Protokoll berechnen. Abbildung A.2 zeigt exemplarisch die Berechnung eines *Requires Protocols* aus einem *Provides Protocol* und SEFFs anhand endlicher Automaten.

Kurz skizziert erfolgt eine Erweiterung des angebotenen Protokolls. Dazu wird das angebotene Protokoll als Grundlage genommen und für jede Transition dieses Automaten wird der Automat des SEFFs eingefügt, der die Realisierung der Methode aus der Beschriftung der Kante enthält. Die Endzustände des SEFFs werden dabei entfernt. Um die aus dem SEFF stammenden Teilautomaten in das ursprüngliche angebotene Protokoll einzufügen, werden zwei ϵ-Transitionen eingefügt: eine führt vom ursprünglichen Startzustand der Transition aus dem angebotenem Protokoll in den ursprünglichen Startzustand des SEFF-Automaten. Eine weitere ϵ-Transition führt vom ursprünglichen Endzustand des SEFF-Automaten in den ursprünglichen Folgezustand der ersetzten Transition des angebotenen Protokolls.

Wird dieses Prozedere für jede Transition des angebotenen Protokolls durchgeführt, erhält man ein benötigtes Protokoll der Komponente. An dieser Stelle soll nicht näher auf die Berechnung des benötigten Protokolls eingegangen, sondern lediglich kurz skizziert werden, dass eine Möglichkeit besteht, das benötigte Protokoll zu berechnen.

Daraus ergibt sich die Notwendigkeit, dass, sofern benötigte Protokolle manuell spezifiziert werden, diese konsistent zu einer berechneten Form sein müssen. Üblicherweise wird man jedoch auf eine manuelle Spezifikation des benötigten Protokolls verzichten und dieses berechnen lassen. Algorithmen zur Berechnung von benötigten Protokollen sind nicht Teil des Komponentenmodells.

A.2.2. Berechnung der SEFFs von Composite Components

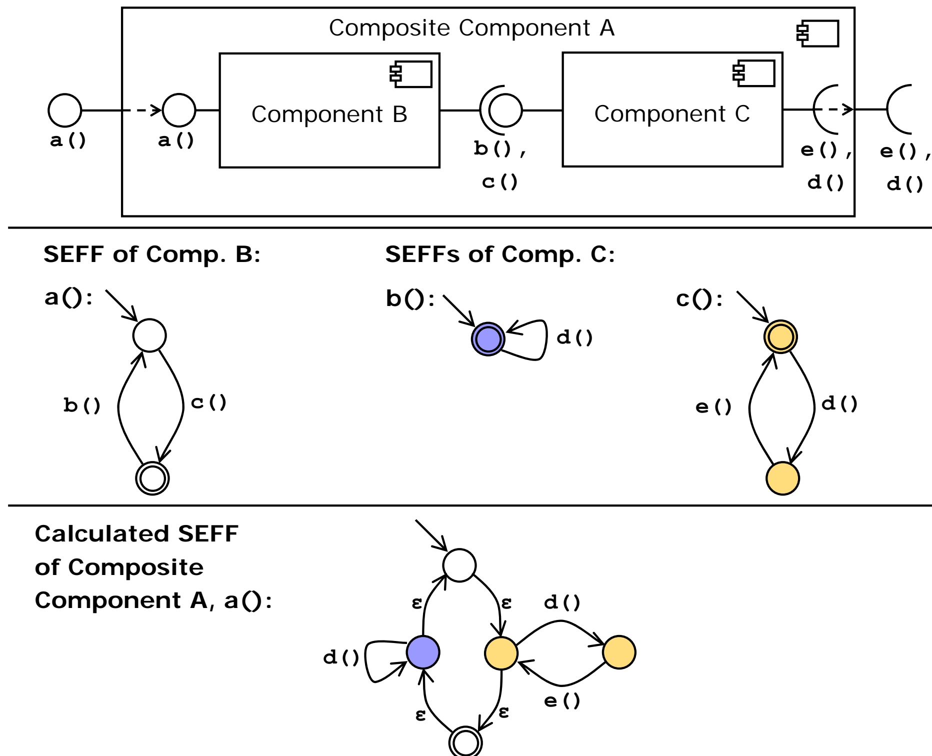


Abbildung A.3.: Beispiel der Berechnung eines SEFFs für eine *Composite Component* bei endlichen Automaten

In Kapitel 2.9 wurde bereits dargelegt, weshalb *Composite Components* selbst keine SEFFs besitzen. Dennoch lässt sich für *Composite Components* ein SEFF berechnen. Dieser ergibt sich aus den zusammengesetzten SEFFs innenliegender Komponenten. Entsprechend ist ein solcher SEFF nur berechenbar, wenn auch für alle innenliegenden Komponenten, einschließlich innerer *Composite Components*, SEFFs vorliegen. Liegen für alle inneren (auch rekursiv absteigend) Komponenten vollständig *Basic Components* und somit auch SEFFs vor, lassen sich die SEFFs höher liegender *Composite Components* rekursiv berechnen. Damit wird dann schließlich auch der SEFF der zu betrachtenden Composite Component berechenbar. Bei einem voll spezifizierten Modell (siehe auch *Implementation-Type* in Kapitel 2.10) verweisen letztlich alle *Composite Components* auf *Basic Components*.

Abbildung A.3 zeigt beispielhaft die Berechnung von SEFFs für eine *Composite Component* aus den SEFFs zweier Sub-Komponenten. Auch hier sind die SEFFs wieder als endliche Automaten dargestellt. Ähnlich der Vorgehensweise bei der Berechnung des *Requires Protocols* erfolgt auch hier eine Ersetzung von Transitionen. Zusammengefasst führen all jene Methodenaufrufe, die nicht aus einer *Composite Component* her-

aus gehen, zur Ersetzung der Transition, die den Methodenaufruf repräsentiert, mit dem SEFF, der dem Methodenaufruf entspricht. Ebenso wie beim Protokoll werden ϵ-Transitionen eingeführt, die in den Startzustand führen und vom Endzustand wieder zurück. Auch der Endzustand des eingefügten SEFFs wird entfernt.

Im Beispiel bietet „Composite Component A“ mit `a()` genau eine Methode an, die durch „Component B“ implementiert wird. „Component B“ delegiert die Aufrufe von `b()` und `c()` auf „Component C“. Diese Komponente benötigt die Methoden `e()` und `d()`. Da `e()` und `d()` nicht von einer inneren Komponente angeboten werden, führen diese Methodenaufrufe aus „Composite Component A“ heraus. Zusätzlich sind die SEFFs zu den Methodenaufrufen `a()`, `b()` und `c()` angegeben, also jenen Methoden, die durch innere Komponenten angeboten werden.

Der berechnete SEFF für „Composite Component A“ und die Methode `a()` ergibt sich entsprechend dem oben skizzierten Verfahren, zunächst aus dem SEFF für `a()` aus „Component B“. In diesem SEFF werden die Aufrufe von `b()` und `c()` durch die entsprechenden SEFFs aus „Component C“ ersetzt (dunkelgrau bzw. blau eingefärbte Zustände resultieren aus dem SEFF von `b()`; hellgrau bzw. gelb eingefärbte Zustände resultieren aus dem SEFF von `c()`).

Da die Berechnung von SEFFs für *Composite Components* damit problemlos und effizient möglich ist, unterstützt das Komponentenmodell für zusammengesetzte Komponenten nur berechnete SEFFs. Diese haben den Vorteil automatisch konsistent mit den inneren SEFFs einer Komponente gehalten werden zu können.

A.3. Erstellungsanweisungen

Zur Vereinfachung der Erstellung eigener neuer Versionen eines GMF basierten Editors, werden an dieser Stelle gesammelte Erfahrungen und Hinweise zur Erstellung aufgeführt. Die Anweisungen beziehen sich im Allgemeinen auf GMF Version 1.0M6. Die Anweisungen sind nicht vollständig, sondern als Ergänzung zu den im folgenden genannten Tutorials zu sehen.

Die Erstellung des GMF basierten Editors orientiert sich grundlegend an den Tutorials aus dem GMF Wiki[1].

Für die Anwendung von EMF auf das Komponentenmodell eignen sich die offiziellen Tutorials[2].

A.3.1. EMF

- Damit EMF das `id`-Feld aus der `EClass Identifier` zur Referenzierung bei der Serialisierung nutzt, muss die Eigenschaft „ID“ für dieses ECORE-Attribut auf „true“ gesetzt werden.

- In der von EMF als Teil des Domänenmodells generierten Klasse `IdentifierModel.impl.IdentifierImpl.java` muss der Konstruktor modifi-

[1] http://wiki.eclipse.org/index.php/GMF_Tutorial,
http://wiki.eclipse.org/index.php/GMF_Tutorial_Part_2
und http://wiki.eclipse.org/index.php/GMF_Tutorial_Part_3
[2] http://www.eclipse.org/emf/docs.php?doc=tutorials/clibmod/clibmod.html und
http://www.eclipse.org/emf/docs.php?doc=tutorials/slibmod/slibmod.html

ziert werden, damit neue Instanzen von Identifier automatisch eine UUID (ähnlich der GUID) erzeugen:

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated not
 */
protected IdentifierImpl() {
  super();
  setId(org.eclipse.emf.ecore.util.EcoreUtil.generateUUID());
}
```

- Bei der Erstellung des Modell-Codes für Protokolle und SEFFs erzeugt EMF fehlerhafte Import-Anweisungen. Durch löschen des fehlerhaften Namespace-Präfixes und anschließenden Imports der von Eclipse vorgeschlagenen Namespaces lässt sich der Fehler beheben. (Andere dieser Fehler ließen sich durch Umbenennung der Diagramme und Namespaces in RSA beheben, so dass der generierte Quellcode direkt kompilierbar war.)

A.3.2. GMF

- Folgt man den Vorgaben der GMF-Wizzards, gelingt es GMF nicht, fehlerfreien Quellcode zu produzieren – offenbar ist das Komponentenmodell hierfür zu komplex. In den Wizzards sollte man zunächst nur für Entitäten und das `EntityName` Attribut GMF-Elemente erzeugen lassen. Nicht jedoch für den Rest. Faktisch wird damit fast jegliche Erzeugung über die Wizzards deaktiviert. Die auf diese Weise erzeugten Transformationsanweisungen bleiben jedoch deutlich übersichtlicher und vor allem kompilierbar. Ergänzungen folgen später besser Stück für Stück manuell.

- **gmfgraph**-Wizzard

  Wie bereits beschrieben, sollten bis auf die Elemente für Entitäten und `EntityName` alle Haken deaktiviert werden. Grundsätzlich sind alle verbleibenden Entitäten als Nodes zu deklarieren – mit Ausnahme von: Provided Role, Required Role, Assembly Connector, Provided Delegation Connector, Required Delegation Connector und Non Calculating Resource.

- **gmftool**-Wizzard

  analog zu gmfgraph

- **gmfmap**-Wizzard

  - Im Mapping Wizzard unbedingt gmfgraph zum Komponentenmodell auswählen, NICHT das vorausgewählte Standard-GMFgraph. Sonst erfolgt die graphische Darstellung nur über Standard-Figures.

- Beim Anlegen von mehreren Plugins (Sichten) darauf achten, dass das richtige Domänenmodell (aus dem richtigen Datei-Ordner) gewählt wird. Auf Grund der notwendigen Duplizierung des Domänenmodells stehen mehrere zur Auswahl.
- Root-Element: „_Factory“
- Mapping von Nodes und Links: Wie bereits bei gmfgraph und gmftool hier bis auf die Kern-Entitäten und als Assoziationsklassen zu realisierenden Links keine weiteren Elemente bestehen lassen – löschen. Zuordnung zu Node und Link, wie bereits in gmfgraph und gmftool.

• **gmfmap**

- Im vom Wizzard erzeugten Mapping müssen alle Zuordnungen überprüft werden. Ein Link sollte anschließend aussehen wie:

```
Link Mapping (Bsp. ProvidedRole)
- Containment Feature: EReference entities__Factory
- Element: EClass ProvidedRole
- Source Feature: EReference providingRole__AssemblyConnector
- Target Feature: EReference requiringRole__AssemlyConnector
- Diagram Link: Connection ProvidedRoleLink
- Creation Tool: Creation Tool ProvidedRole
```

- Nodes sollten wie folgt realisiert sein:

```
- Top Node Reference (Bsp. Interface)
- Containment Feature: EReference entities__Factory
- Node Mapping:
  - Element: EClass Interface
  - Diagram Node: Node InterfaceNode
  - Tool: Creation Tool Interface
```

- Als Label Mapping (unterhalb von Link Mapping bzw. Node Mapping) empfiehlt sich:

```
- Diagram Label: Diagram Label EntityEntityNameLabel
- Edit Pattern: ProvidedRole: {0}
- Features: EAttribute entityName, EAttribute id
- Read Only: false
- View Pattern: ProvidedRole: {0}, {1}
```

- Das Label Mapping bei Entites sollte konsistent so aussehen:

```
- [EntityName] {0} als Edit Pattern
- [EntityName] {0}, {1} als View Pattern
```

- Damit ist `entityName` bearbeitbar. Der Entitäts-Typ wird immer angezeigt, ebenso wie die ID. Bei der Interpretation wird im Übrigen `java.text.MessageFormat` verwendet.

  - Zur Realisierung von Compartments sollte umbedingt die Option `ListLayout=false` in gmfgen für alle Compartments gesetzt werden. Damit sind frei bewegliche Figures möglich. Nur frei bewegliche Compartment-Elemente eignen sich als Ankerpunkte für Links.
  - Die Zuordnung von Tools zu Entitäten in gmfmap ist in jedem Fall manuell zu überprüfen. Das Auto-Mapping des Wizzards schlagen hier fehl.

- Eine einfache Pfeilform für RequiredRole ist über die folgenden TemplatePoints in einer Polyline Decoration „HalfCircle“ mit Koordinaten x/y möglich:

```
- 1,2
- 0,0
- 1,-2
```

- Shortcuts (vgl. GMF Tutorial 2): 'Shortcuts Provided For' und 'Contains Shortcuts To' müssen auf in gmfgen auf „palladiocm“ gesetzt sein, damit Entitäten aus dem gleichen Domänenmodell importiert werden können.
- Zum Import von Entitäten externer Plugins (SEFF, Protocol, Annotation) über Shortcuts sind zusätzlich folgende Endungen vorzusehen: „PcmSeff“, „PcmProtocol“, „PcmAnnotation“.
- Eigene Figures am Beispiel des Interfaces:
  - InterfaceNode hat als graphische Repräsentation einen Circle aus Draw2D: org.eclipse.gmf.runtime.gef.ui.internal.figures.CircleFigure. Diesen unter „Custom Figure“ als „Qualified Class Name“ setzen.
  - In der generierten Klasse `InterfaceEditPart.java` muss der folgende Code-Block daraufhin modifiziert werden:

```
/**
 * @generated not
 */
public CustomInterfaceFigure() {
  super(10); //call of the non-empty super constructor required
  [..]
```

- Nutzung mehrerer Diagramm-Plugins mit dem gleichen Domänenmodell
  - Nur ein Diagramm-Plugin wird dazu genutzt, Modell-Code und Edit-Code zu generieren. Alle anderen Projekte referenzieren diesen Code. Auf diese Weise entstehen keine unnötigen Duplikationen des gleichen Quellcodes. Zudem kann der Aufwand für manuelle Korrekturen minimiert werden.
  - Eine Duplikation von *.ecore-Modellen sowie .genmodel ist für alle zusätzlichen Diagramm-Plugins notwendig, damit GMF das Domänenmodell korrekt referenzieren kann.
  - Je Modell muss eine Anpassung der folgenden Eigenschaften erfolgen:

```
Gen Editor Generator:
- Diagram File Extension
- Model ID
- Package Name Prefix

Gen Editor:
- Action Bar Contributor Class Name
- Class Name

Gen Plugin:
- Activator Class Name
- ID
- Name

Gen Diagram:
- no manual changes
```

Zusätzlich werden einige abhängige Eigenschaften automatisch gesetzt.

– Anpassung der generierten Wizzards eines Diagramm-Plugins bei mehr als einem Diagramm-Plugin – diese ermöglichen eine einfachere Unterscheidung (hier für „allocation"):
`PalladioCMCreationWizard.java`:

```
/**
 * @generated not
 */
public void init(IWorkbench workbench, IStructuredSelection selection) {
  super.init(workbench, selection);
  setWindowTitle("New PalladioCMallocation Diagram"); //$NON-NLS-1$
  [..]
```

`PalladioCMCreationWizardPage.java`:

```
/**
 * @generated not
 */
public PalladioCMCreationWizardPage(IWorkbench workbench,
    StructuredSelection selection) {
  super("CreationWizardPage", workbench, selection); //$NON-NLS-1$
  setTitle("Create PalladioCMallocation Diagram"); //$NON-NLS-1$
  setDescription("Create a new PalladioCMallocation diagram."); //$NON-NLS-1$
}
```

- Wurden umfangreiche Änderungen an den Domänenmodellen vorgenommen, ist es in den meisten Fällen einfacher, wenn der generierte Quellcode komplett gelöscht und anschließend neu generiert wird. Die automatischen Merges versagen in diesen Fällen zumeist.

- Ergänzend sollten alle in Kapitel 5 genannten Einschränkungen und Fehlerursachen beachtet werden.

A.4. UML-Diagramme

Im Folgenden werden die UML2-Diagramme aus der Modellierung des Komponenten-Meta-Modells dargestellt. Insbesondere die Formatierung von Constraints wurde beim Export in Mitleidenschaft gezogen. Die hier angegebenen Diagramme sind daher zusätzlich in der elektronischen Version dieser Diplomarbeit zu finden – dort dann auch in der Größe frei skalierbar.

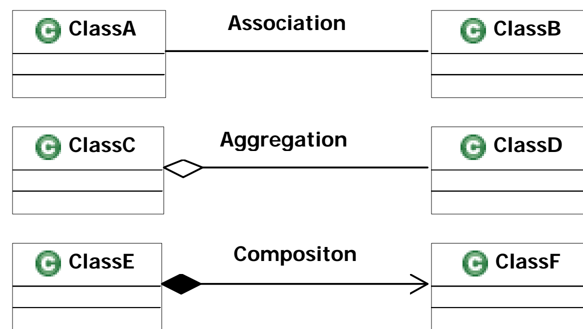


Abbildung A.4.: UML2-Notation: Assoziation (ungerichtet), Aggregation (ungerichtet), Komposition (gerichtet) – von oben nach unten

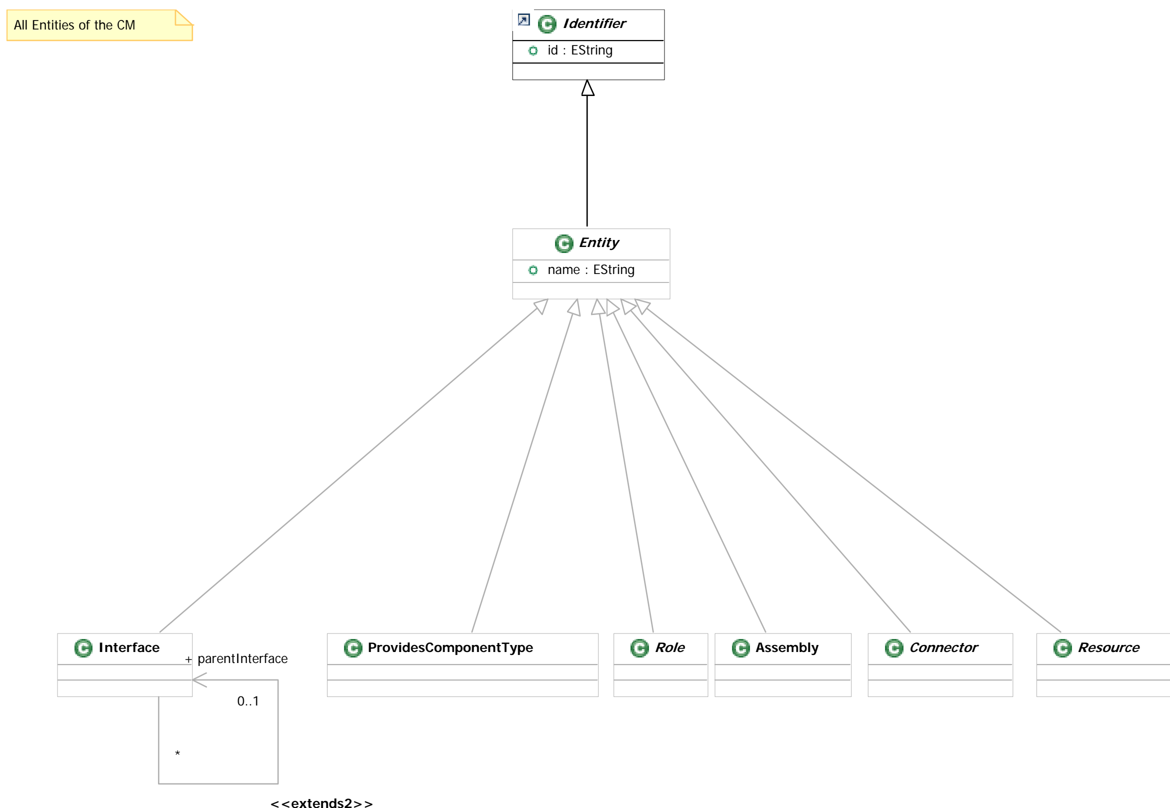


Abbildung A.5.: UML: Entitäten

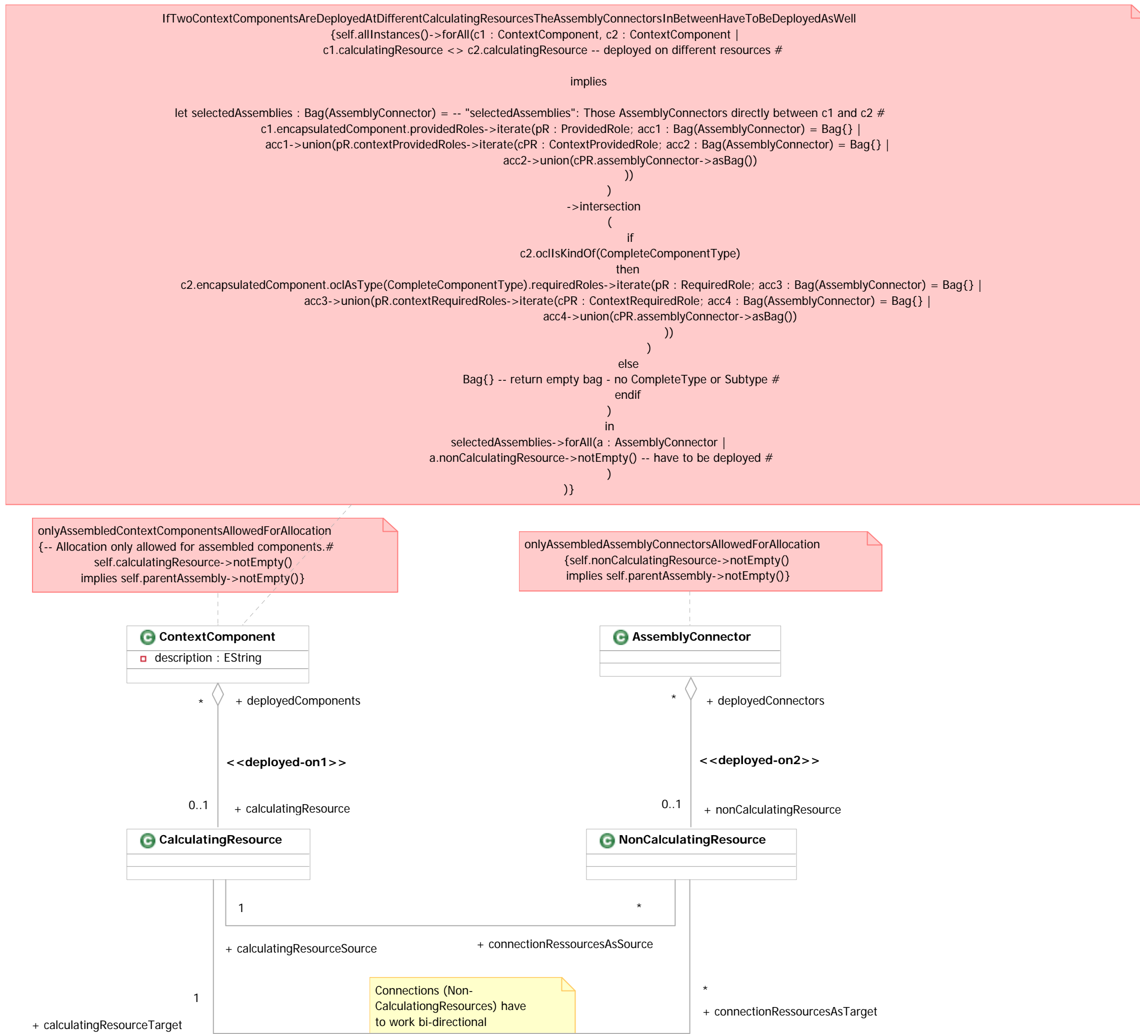


Abbildung A.6.: UML: Allokation von Kontext-Komponenten und Assembly Konnektoren auf Ressourcen

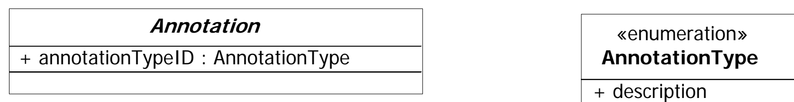


Abbildung A.7.: UML: Annotation

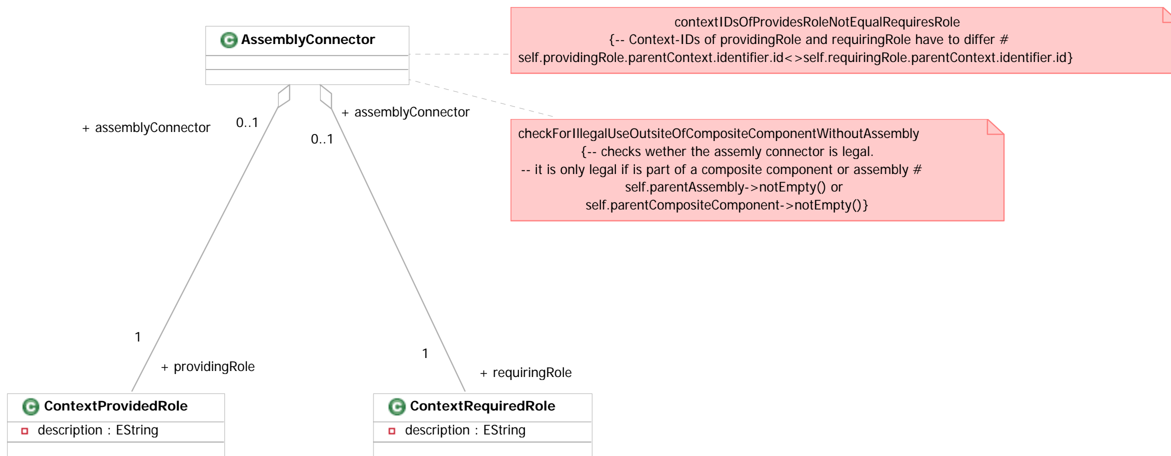


Abbildung A.8.: UML: Assembly Konnektor

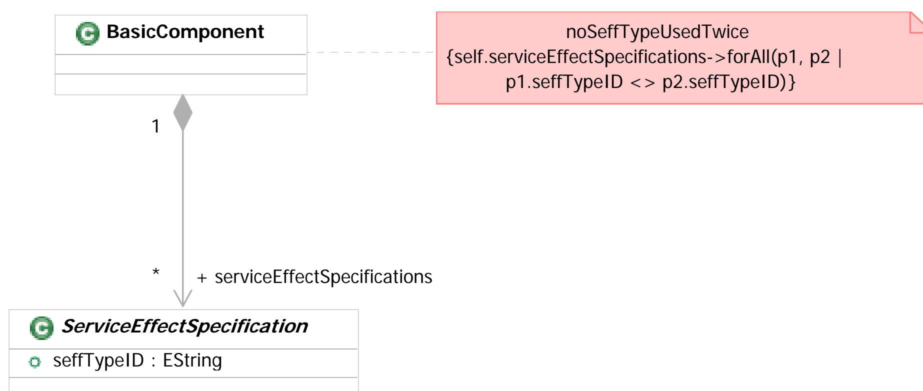


Abbildung A.9.: UML: *Basic Components* und SEFFs

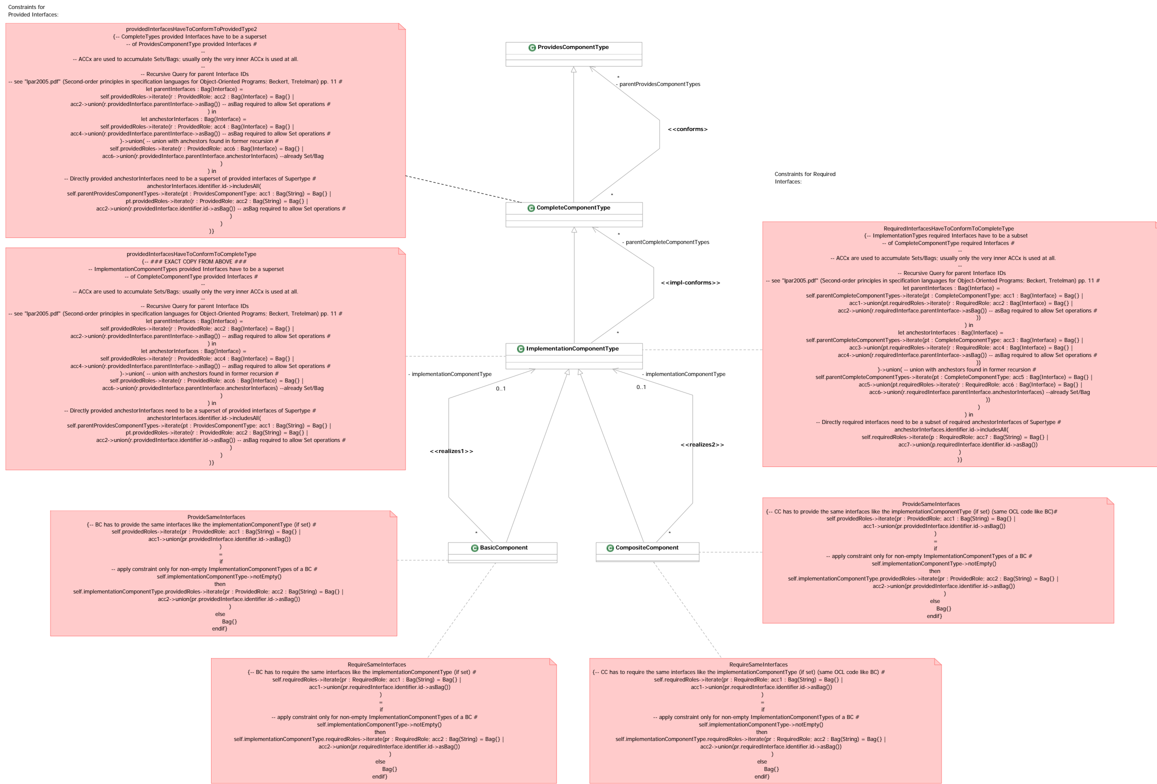


Abbildung A.10.: UML: Hierarchie der Komponenten-Typen

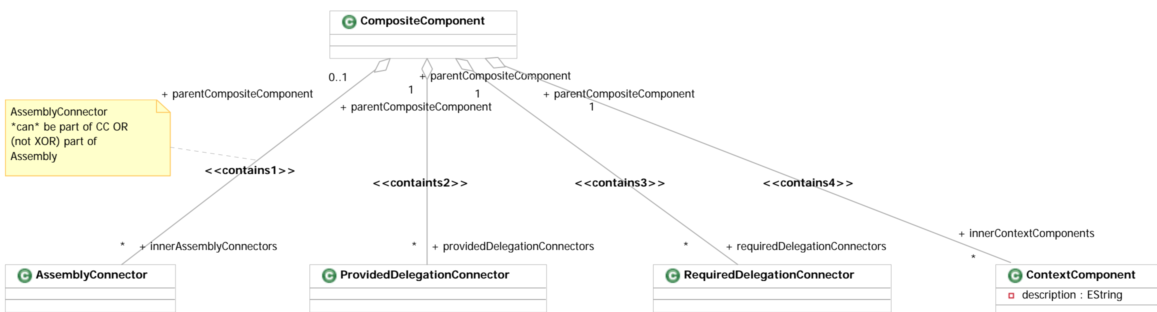


Abbildung A.11.: UML: *Composite Component*

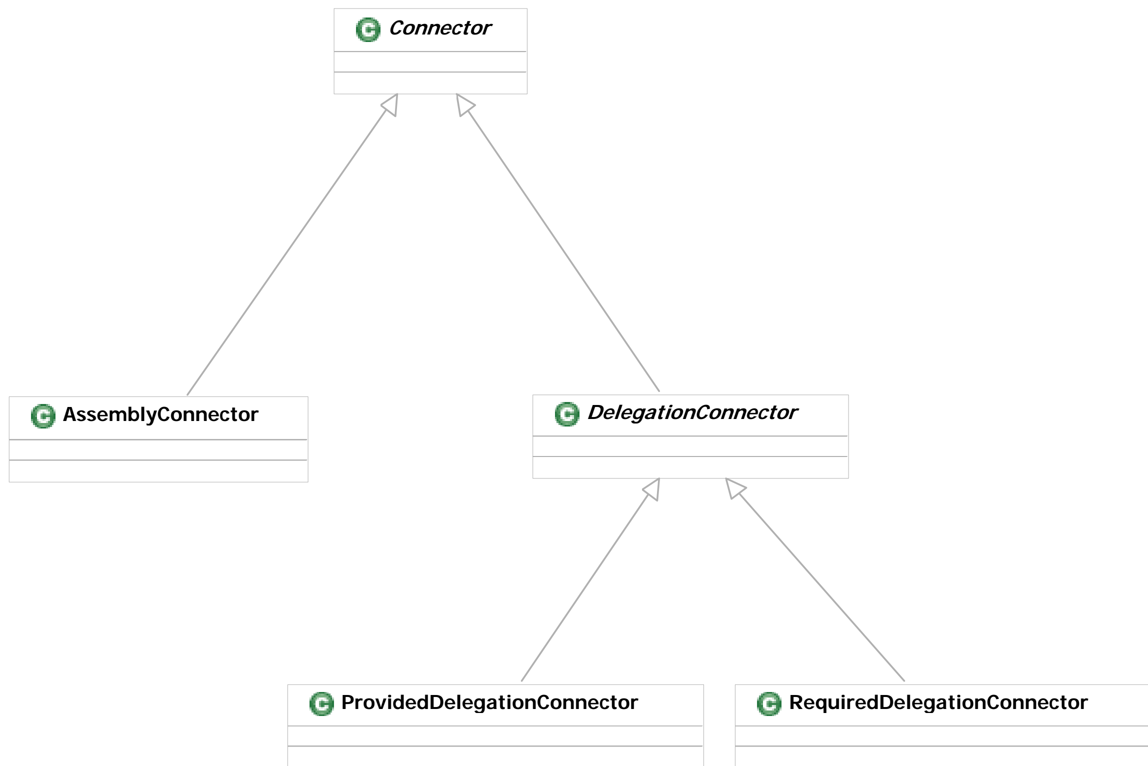


Abbildung A.12.: UML: Vererbungshierarchie der Konnektoren

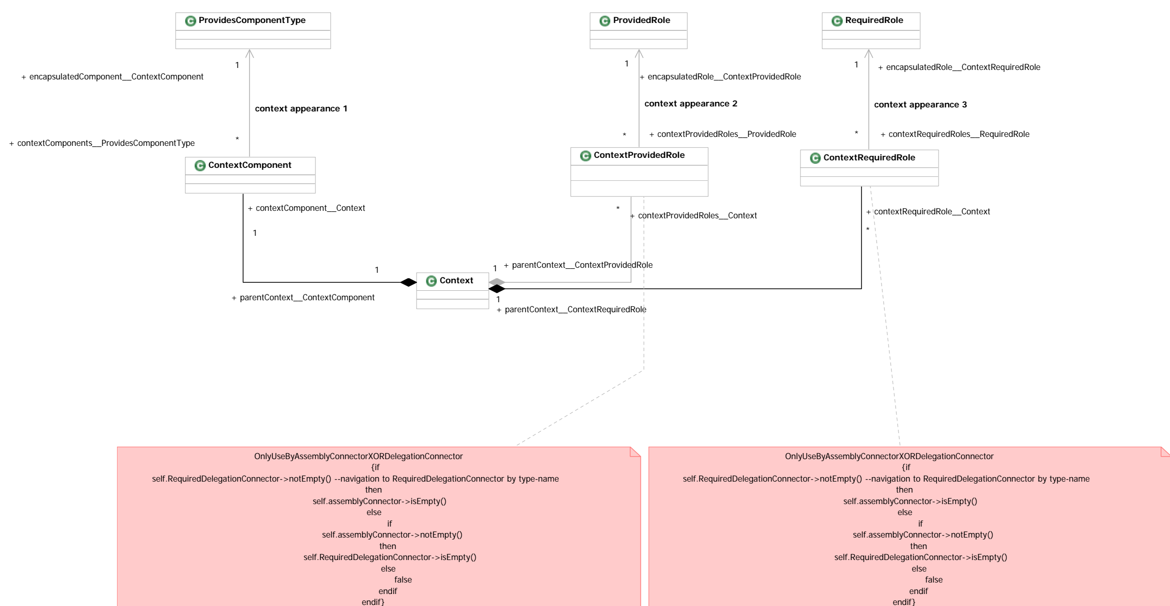


Abbildung A.13.: UML: Kontext-Komponenten und Kontext-Rollen des Konzext-Konzepts

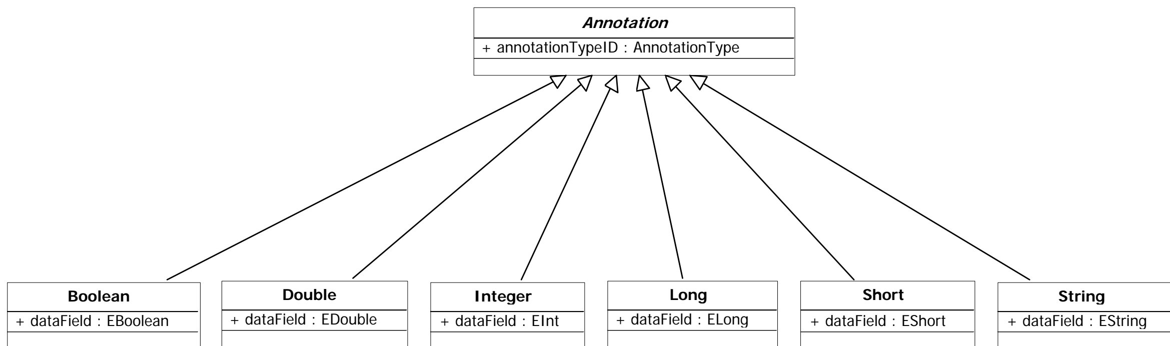


Abbildung A.14.: UML: Standard Datentypen für Annotationen

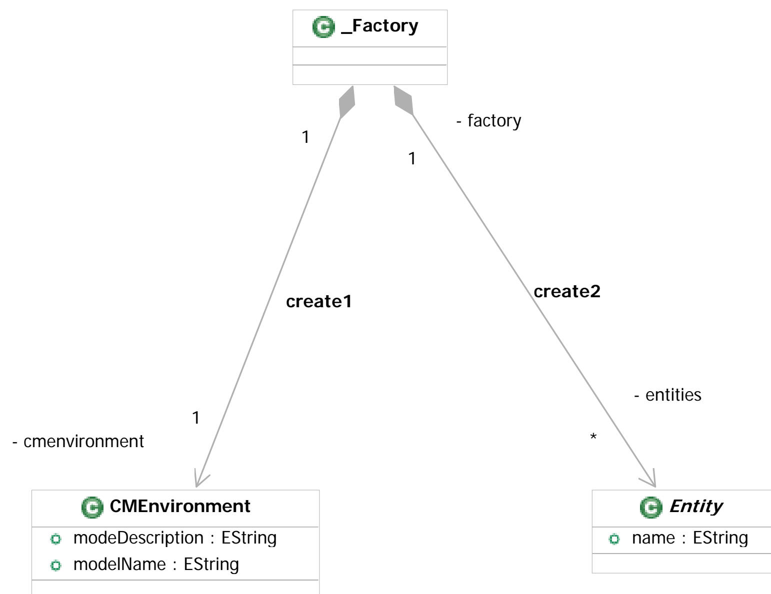


Abbildung A.15.: UML: Factory des Komponentenmodells

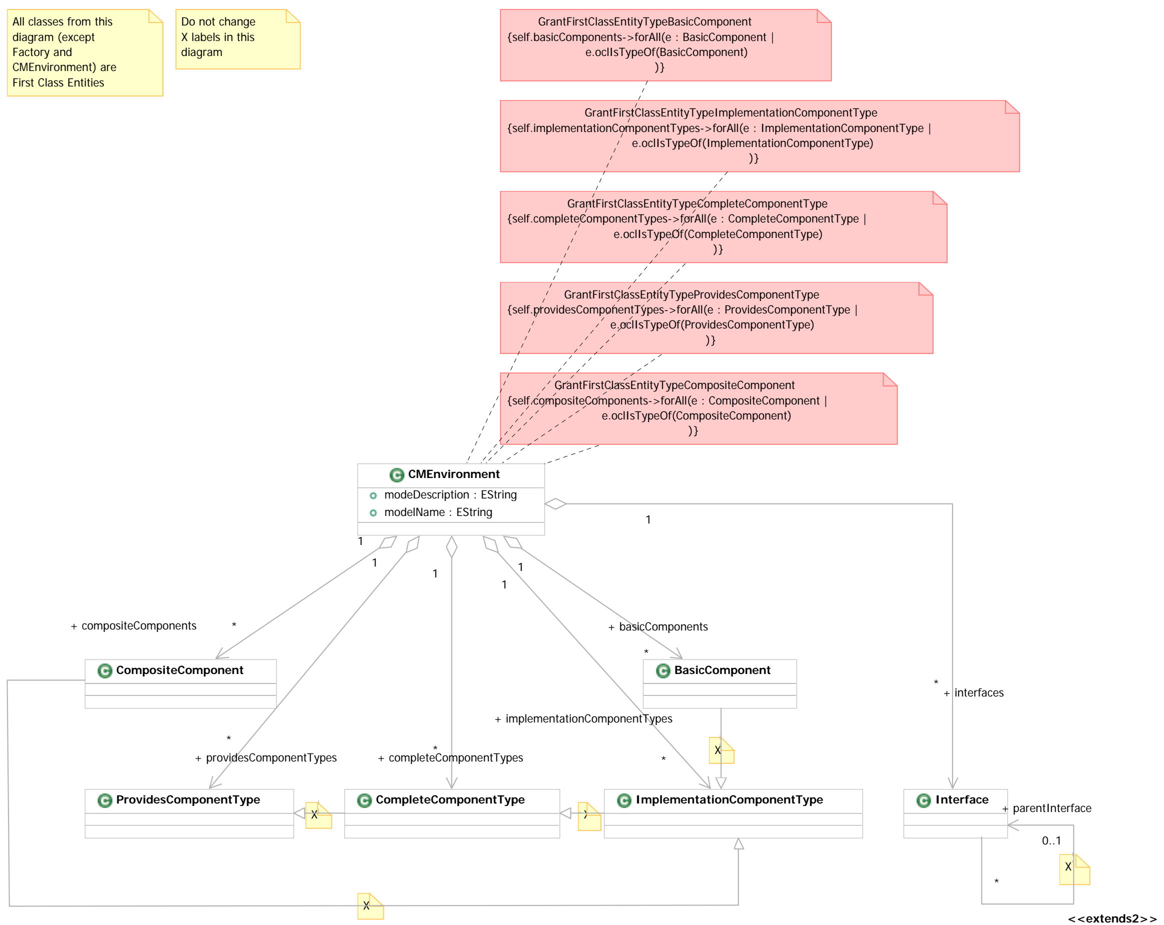


Abbildung A.16.: UML: First Class Entities

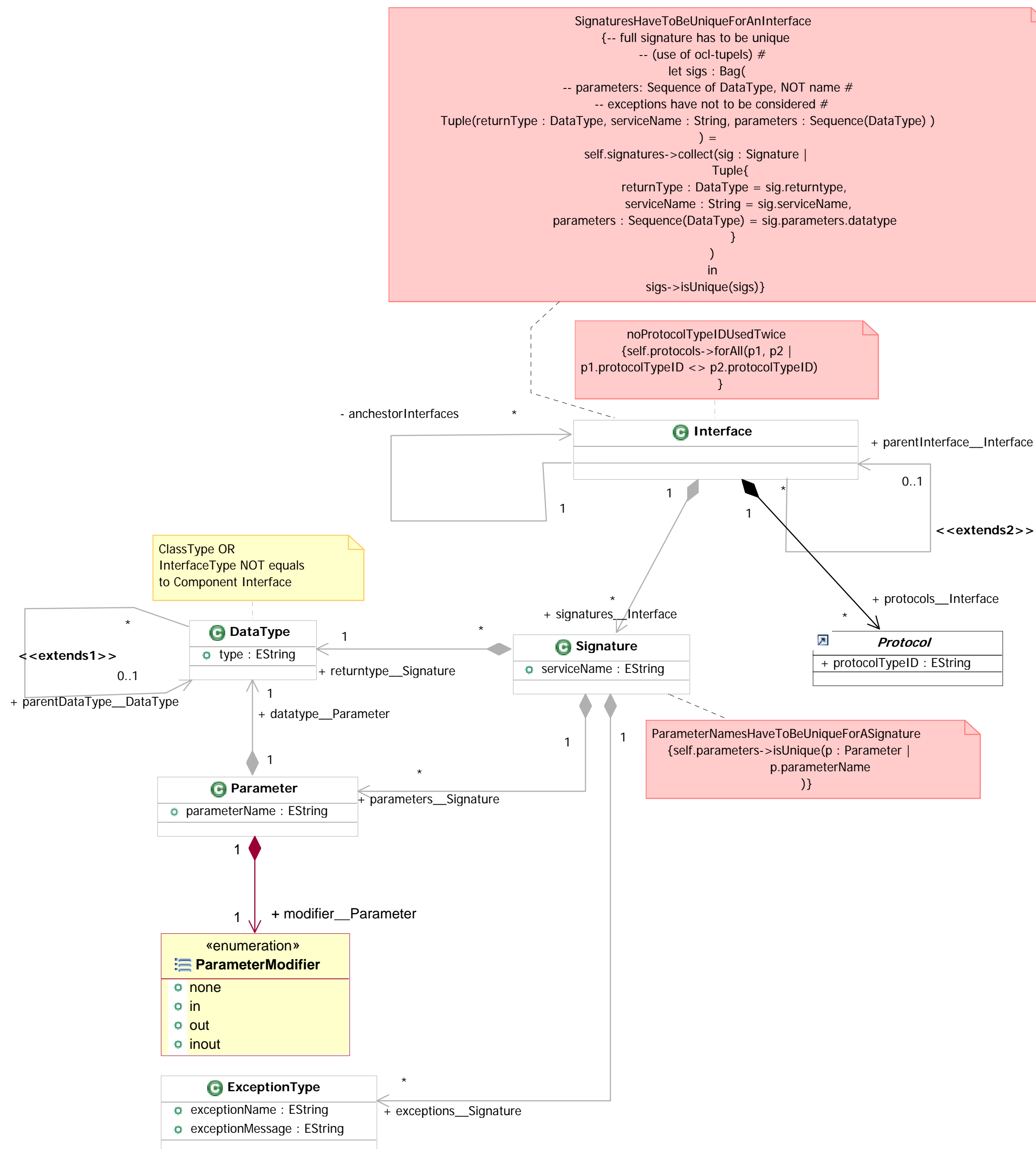


Abbildung A.17.: UML: Schnittstellen und Signaturen

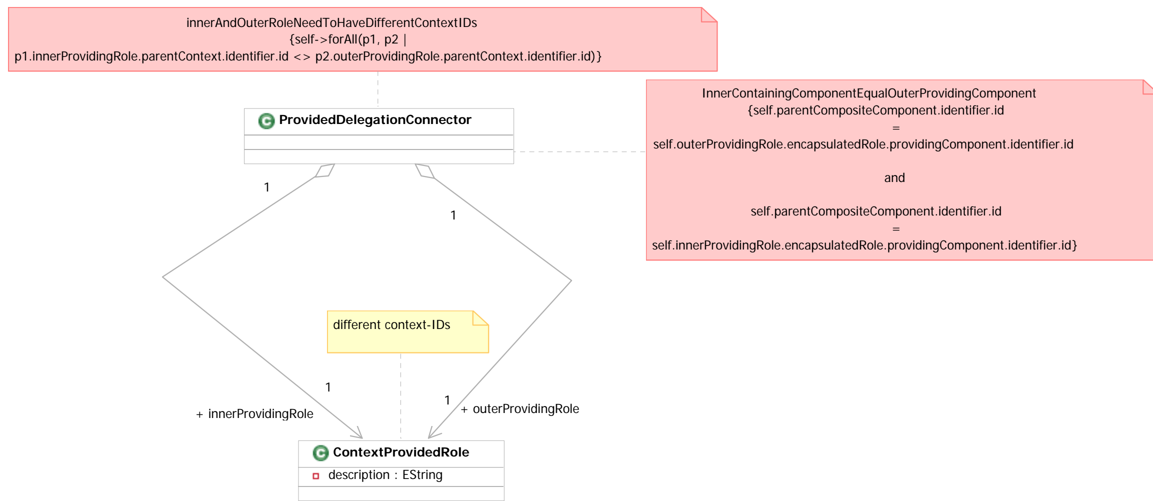


Abbildung A.18.: UML: Angebotene Delegations-Konnektoren

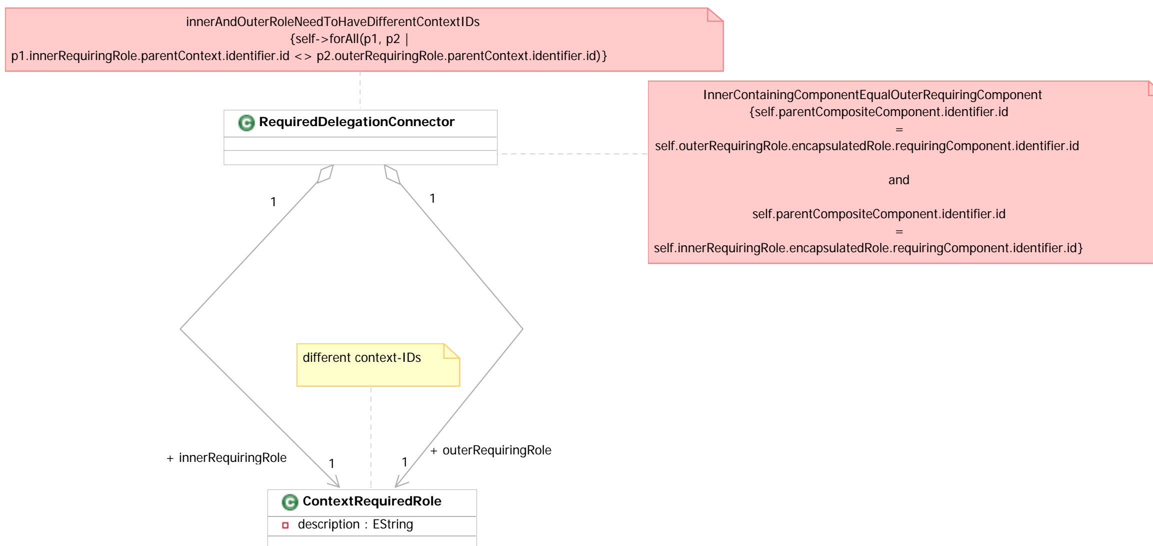


Abbildung A.19.: UML: Benötigte Delegations-Konnektoren

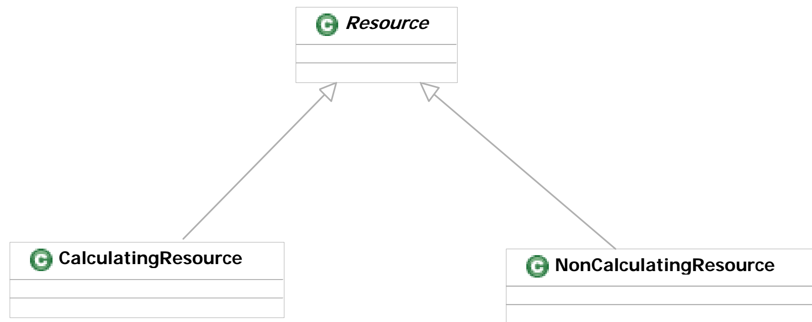


Abbildung A.20.: UML: Vererbungshierarchie von Ressourcen

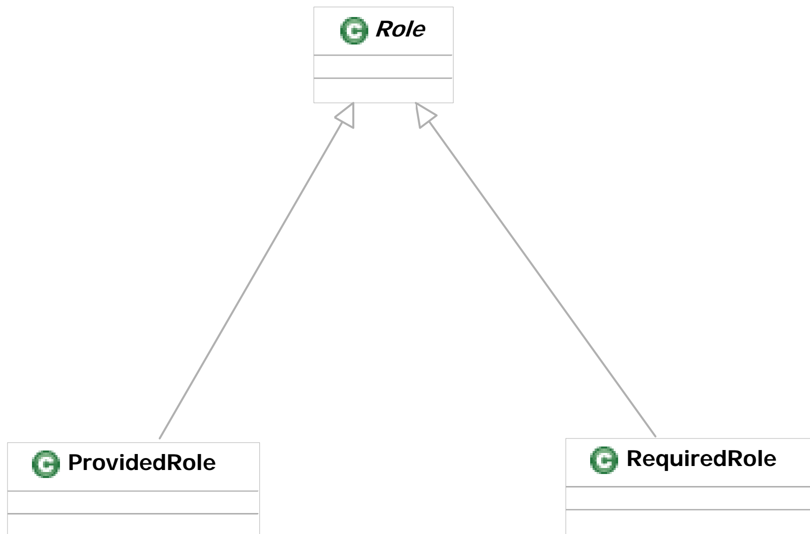


Abbildung A.21.: UML: Vererbungshierarchie von Rollen

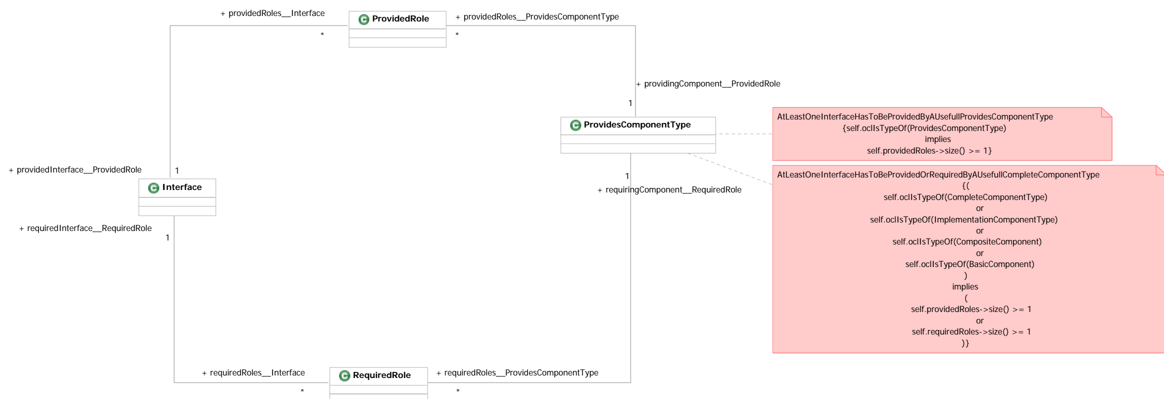


Abbildung A.22.: UML: Definition von Rollen

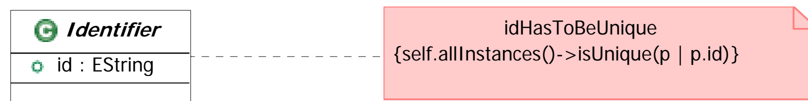


Abbildung A.23.: UML: IdentifierModel: Identifier

A.5. Abbildungen: Alternative Identifier-Modellierung

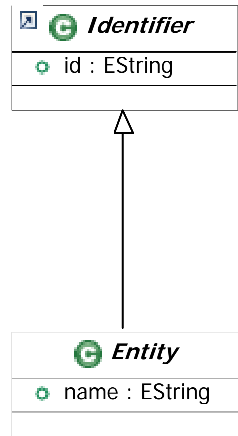


Abbildung A.24.: Identifier- und Entity-Vererbungsstruktur

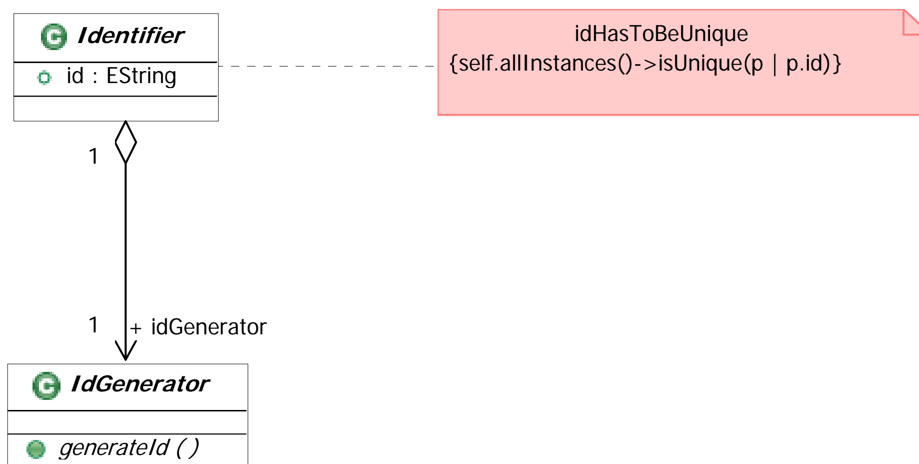


Abbildung A.25.: Identifier und IdGenerator

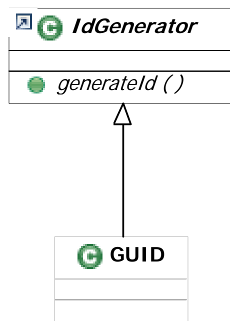


Abbildung A.26.: GUID-Vererbungsstruktur

Problems | Javadoc | Declaration | Properties

Property	Value
Id	BEAE36F4-D834-D2DA-7DDA-E55875EC8437
Id Generator	GUID
Name	PRb
Provided Interface	Interface Ia
Providing Component	

Abbildung A.27.: Bildschirmfoto: *Properties-View* im Editor

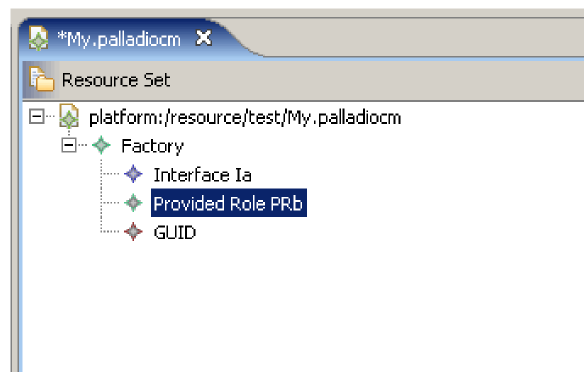


Abbildung A.28.: Bildschirmfoto: Baumansicht des generierten Editors

A.6. Glossar

Sub-Typ Beim Sub-Typ handelt es sich um eine Relation zwischen zwei Komponenten *oder* zwei Schnittstellen. Gilt diese Relation, so kann ein Sub-Typ für den „Super-Typ“ verwendet werden. Damit ist der Sub-Typ eine Spezialisierung des „Super-Typs“. Siehe hierzu auch Kapitel 2.11 auf Seite 39. Dort wird ebenfalls auf Co- und Contra-Varianz im Zusammenhang mit Komponenten eingegangen.

Ein Komponenten-Typ (siehe Kapitel 2.10.2) der in einer *Conforms*-Beziehung (vgl. 2.10.3) zu einem anderen Komponenten-Typ steht, ist Sub-Typ dieses Komponenten-Typs.

Implementierung Wird der Begriff Implementierung verwendet, so ist eine Binärinstanz einer Komponente / Schnittstelle oder eine programmiersprachliche Repräsentation einer Komponente / Schnittstelle gemeint. Wie im Bereich der objektorientierten Programmierung (siehe etwas [30], S. 203ff) gibt es von Komponenten-“Vorlagen“ Implementierungen – analog zu OO-*Interfaces* und Klassen. Präzise betrachtet kann es zu einer Ausprägung eines Komponenten-Typs `0..*` Implementierungen geben.

Implementierungen sind nicht mit dem *Implementation Component Type* zu verwechseln.

RSA RSA ist die Abkürzung für *IBM Rational Software Architect*. Dieses Tool wird zur Modellierung von UML2-Modellen sowie zur Erstellung von OCL-Constraints verwendet. Über die Exportfunktionen stehen die Möglichkeiten offen, ECORE-Modelle (Serialisierungsformat von EMF) oder UML2-Modelle (Serialisierungsformat der OMG für UML2) zu schreiben.

EMF EMF ist das *Eclipse Modeling Framework* der Eclipse Foundation. Für die Diplomarbeit wird über EMF die zentrale Transformationsarbeit zur Generierung eines Java-Modells des Komponentenmodells abgewickelt. Zudem dient ein über EMF generierbarer Editor zur einfachen Modellierung von Instanzen des Komponentenmodells. Als Serialisierungsformat für EMF dient ECORE.

Eine Einführung in EMF findet sich in [19], Kapitel 2: Einführung in EMF.

GEF GEF ist das *Graphical Editing Framework* der Eclipse Foundation, einem Framework zur Erstellung von graphischen Editoren, die auf Eclipse basieren.

GMF Hinter GMF verbirgt sich das *Graphical Modeling Framework*. Es versteht sich als generative Brücke zwischen EMF und GEF. Mit Hilfe von GMF lassen sich aus EMF heraus graphische Editoren generieren. Die Transformationsvorgänge sind dabei stark konfigurierbar gehalten, um erzeugte Editoren eigenen Bedürfnissen anpassen zu können. GMF befindet sich im Entwicklungsstatus.

JET Die *Eclipse Java Emitter Templates* (JET) sind ein Open Source Werkzeug zur Generierung von Quellcode, das unter anderem im Eclipse Modeling Framework eingesetzt wird. Es ist ähnlich den *Java Server Pages*, aber mächtiger und flexibler angelegt um auch Java-Code, SQL, XML und andere Sprachen erzeugen zu können.

Abbildungsverzeichnis

1.1. Meta-Modell-Ebenen der UML 2.0 und des Komponenten-Meta-Modells im Vergleich 14

2.1. Einfache Komponente mit einer angebotenen und zwei benötigten Schnittstellen 21
2.2. Angebotene und benötigte Schnittstellen mit ihren dazugehörigen Rollen in UML2-Notation, sowie anbietende / benötigende Komponente (in abkürzender Schreibweise) 22
2.3. Angebotene und benötigte Schnittstellen mit ihren dazugehörigen Rollen in UML2-Notation (als Schreibweise mit Stereotypen). Semantisch äquivalent zu Abbildung 2.2. 22
2.4. Eine Komponente wird über Assembly Konnektoren mit zwei weiteren Komponenten verbunden 24
2.5. Eine zusammengesetzte Komponente mit inneren Komponenten 25
2.6. Beispielschnittstellen mit Signaturlisten und Protokollinformationen in Form von endlichen Automaten 27
2.7. Service Effekt Spezifikation als endlicher Automat zu beispielhaftem Pseudo-Quellcode 30
2.8. Typ-Hierarchie: *Provided Type*, *Complete Type* und *Implementation Type* (nach [10], S. 18) 34
2.9. Komponenten Typ-Ebenen: Vererbungs- und Konformitätsbeziehungen 36
2.10. Spezialisierung eines *Provided Type* zu zwei *Complete Types* (nach [10], S. 17) 38
2.11. Szenario in dem Komponente X („Component X“) in einer bestehenden Komponentenarchitektur substituiert werden soll 40
2.12. Trivial-Protokoll als FSM dargestellt: Dienstaufrufe sind in beliebiger Reihenfolge möglich 41
2.13. Illegale und legale Allokation von Komponenten 45
2.14. Allokation einer Assembly in einer Systemumgebung (nach [10], S. 26) . 46
2.15. Transformation einer Assembly in eine *Composite Component* (aus Sicht eines *System Deployers*) 48
2.16. Komponentenhierarchie mit Einordnung des Kontexts (nach [10], S. 33) 48
2.17. Unterscheidung von Komponenten-Typ und Komponenten-Kontext . . 49
2.18. Komponente-Typ „Component A“ in zwei unterschiedlichen Kontexten 50
2.19. Invalides Modellkonstrukt: Konsistenzverletzung 55
2.20. Beispiel: Annotation von validitätsrelevanten Informationen 56
2.21. Komponente mit zwei angebotenen Schnittstellen und den darauf spezifizierten Protokollen. 57

2.22. Die gleiche Schnittstelle wird zweifach angeboten; eine Schnittstelle wird zweifach verwendet 58
2.23. Die gleiche Schnittstelle wird zweifach benötigt; eine Schnittstelle wird zweifach delegiert 60
2.24. Rekursive Definition einer *Composite Component* aus unterschiedlichen Sichten 61
2.25. Beispiel: Notwendigkeit für einen Stimulus-Response-Mechanismus . . . 62

3.1. Originär angestrebter Entwicklungsprozess 71
3.2. Iterativer Modellierungsprozess 71
3.3. Modell-Hierarchie (nicht abschließende Aufzählung von Sub-Modellen) . 76
3.4. Protokoll-Modell 76
3.5. SEFF-Modell 77
3.6. FSM-Wrapper (vereinfacht) 77
3.7. Alternative zur Modellierung der Komponenten-Typ-Hierarchie 83

4.1. Generierter EMF Editor mit beispielhafter Modell-Instanz 109
4.2. Testfall für das Komponentenmodell 114

5.1. Übersicht über den Verarbeitungsprozess von GMF (nach dem *Dashboard* des GMF-Plugins) 120
5.2. *Repository* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8]) 122
5.3. *Assembly* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8]) 123
5.4. *Allocation* Sicht auf eine exemplarische Instanz des Komponentenmodells (nach [8]) 123
5.5. Erstellung einer Kontext-Instanz eines Komponenten-Typs 128

6.1. Zuordnung von Transformationsschritten zu Werkzeugen 140

A.1. Vollständige Hierarchie von Komponentenarten 147
A.2. Beispiel der Berechnung eines *Requires Protocols* aus einem *Provides Protocol* und SEFFs 148
A.3. Beispiel der Berechnung eines SEFFs für eine *Composite Component* bei endlichen Automaten 149
A.4. UML2-Notation: Assoziation (ungerichtet), Aggregation (ungerichtet), Komposition (gerichtet) – von oben nach unten 155
A.5. UML: Entitäten 155
A.6. UML: Allokation von Kontext-Komponenten und Assembly Konnektoren auf Ressourcen 156
A.7. UML: Annotation 156
A.8. UML: Assembly Konnektor 157
A.9. UML: *Basic Components* und SEFFs 157
A.10. UML: Hierarchie der Komponenten-Typen 158
A.11. UML: *Composite Component* 158
A.12. UML: Vererbungshierarchie der Konnektoren 159
A.13. UML: Kontext-Komponenten und Kontext-Rollen des Konzext-Konzepts 159
A.14. UML: Standard Datentypen für Annotationen 160

A.15.UML: Factory des Komponentenmodells 160
A.16.UML: First Class Entities 161
A.17.UML: Schnittstellen und Signaturen 162
A.18.UML: Angebotene Delegations-Konnektoren 163
A.19.UML: Benötigte Delegations-Konnektoren 163
A.20.UML: Vererbungshierarchie von Ressourcen 164
A.21.UML: Vererbungshierarchie von Rollen 164
A.22.UML: Definition von Rollen 165
A.23.UML: IdentifierModel: Identifier 165
A.24.Identifier- und Entity-Vererbungsstruktur 166
A.25.Identifier und IdGenerator 166
A.26.GUID-Vererbungsstruktur 167
A.27.Bildschirmfoto: *Properties-View* im Editor 167
A.28.Bildschirmfoto: Baumansicht des generierten Editors 167

Literaturverzeichnis

[1] "Co- and Contra-Variance in Object Oriented Programming," [letztes Abrufdatum 05.01.2006]. [Online]. Available: http://www.csd.uu.se/kurs/oop/ht98/Lectures/D5/pps/Conformance.pps

[2] D. Akehurs, P. Linington, and O. Patrascoiu, "OCL 2.0: Implementing the Standard," Computing Laboratory, University of Kent, Canterbury, UK, Tech. Rep. 12-03, 2003, [letztes Abrufdatum 03.03.2006]. [Online]. Available: http://www.cs.kent.ac.uk/projects/ocl/Documents/OCLReport.pdf

[3] D. Akehurs and O. Patrascoiu, "OCL 2.0 – Implementing the Standard for Multiple Metamodels," Computing Laboratory, University of Kent, Canterbury, UK, Tech. Rep., 2003, [letztes Abrufdatum 03.03.2006, Vorläufige Version]. [Online]. Available: http://www.cs.kent.ac.uk/projects/ocl/Documents/OCL%202.0%20-%20Implementing%20the%20Standard.pdf

[4] Architecture Board ORMSC, "Model Driven Architecture (MDA)," Object Management Group, Tech. Rep., 2001, [letztes Abrufdatum: 26.02.2006] Document number ormsc/2001-07-01. [Online]. Available: http://www.omg.org/docs/ormsc/01-07-01.pdf

[5] C. Ashbacher, *Introduction to Neutrosophic Logic*, 7th ed. Rehoboth: American Research Press, 2002.

[6] D. Bálek and F. Plášil, "Software Connectors: A Hierarchical Model," [letztes Abrufdatum 11.01.2006]. [Online]. Available: http://nenya.ms.mff.cuni.cz/publications/tr2000_2_revised.pdf

[7] ——, "Software Connectors and thier Role in Component Deployment," [letztes Abrufdatum 11.01.2006]. [Online]. Available: http://nenya.ms.mff.cuni.cz/publications/DAIS01.pdf

[8] S. Becker, "Model driven approches to predictable CB software systems," [letztes Abrufdatum 22.04.2006]. [Online]. Available: http://i43pc12.ipd.uni-karlsruhe.de/wiki/Bild:DRRunde190406Steffen.ppt

[9] ——, "The Palladio Component Model," [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://se.informatik.uni-oldenburg.de/pubdb_files/pdf/TechReportComponentModel.pdf

[10] S. Becker and J. Happe, "The Palladio Component Model Meta Model – Concepts and Ideas," [unpublished].

[11] S. Becker, J. Happe, and R. Reussner, “The Palladio Component Model,” [unpublished].

[12] S. Becker, S. Overhage, and R. Reussner, “Classifying software component interoperability errors to support component adaption.” in *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds. Springer, 2004, pp. 68–83.

[13] B. Beckert and K. Trentelman, “Second-Order Principles in Specification Languages for Object-Oriented Programs,” 2005, [letztes Abrufdatum 27.02.2006]. [Online]. Available: http://www.uni-koblenz.de/~beckert/pub/lpar2005.pdf

[14] Borland Software Corporation, “Together 6,” [letztes Abrufdatum 26.02.2006]. [Online]. Available: http://www.borland.com/together

[15] G. Bracha, “Generics in the Java Programming Language,” July 2004, [letztes Anrufdatum: 21.12.2005]. [Online]. Available: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[16] A. D. Brucker and B. Wolff, “A Proposal for a Formal OCL Semantics in Isabelle/HOL,” Albert-Ludwigs-Universität Freiburg, Springer-Verlag Berlin Heidelberg, LNCS 2410, 2002, [letztes Abrufdatum 27.02.2006]. [Online]. Available: http://www.brucker.ch/bibliography/download/2002/ocl_semantic.pdf

[17] ——, “HOL-OCL: Experiences, Consequences and Design Choices,” Albert-Ludwigs-Universität Freiburg, Springer-Verlag Berlin Heidelberg, LNCS 2460, 2002, [letztes Abrufdatum 27.02.2006]. [Online]. Available: http://www.brucker.ch/bibliography/download/2002/holocl_experiences.pdf

[18] E. Bruneton, T. Coupaye, and J. B. Stefanie, “The Fractal Component Model – Specification,” The ObjectWeb Consortium, Tech. Rep., Februar 2004, [letztes Abrufdatum: 08.05.2005]. [Online]. Available: http://fractal.objectweb.org/specification/fractal-specification.pdf

[19] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose, *Eclipse Modeling Framework*, 1st ed. Addison Wesley Professional, Chapter 2. [Online]. Available: http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf

[20] R. Buschermöhle, “Einführung in die MDA,” [letztes Abrufdatum 26.02.2006]. [Online]. Available: http://www.software-kompetenz.de/?5348

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System Of Patterns*. Chichester, West Sussex, England: John Wiley and Sons Ltd., 1996, vol. 1.

[22] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers, “Bridging the MS/DSL Tools and the Eclipse Modeling Framework,” OOPSLA 2005, 2005, aT-LAS Group (INRIA & LINA, University of Nantes).

[23] Carl-Von-Ossietzky Universität, Oldenburg, Department für Informatik, "Projektgruppe Ride.NET," [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://www.Ride.NET.ms/

[24] M. V. Cengarle and A. Knapp, "A formal semantics for OCL 1.4," in *The Unified Modeling Language (UML 2001)*, ser. LNCS, no. 2185. Springer, 2001.

[25] J. Conway and S. Watts, *A Software Engineering Approach to Labview*, 1st ed., Pearson, Ed., New Jersey, 2003.

[26] P. Dadam, *Verteilte Datenbanken und Client/server-systeme.: Grundlagen, Konzepte und Realisierungsformen.* Berlin: Springer-Verlag, 1996.

[27] Distributed Systems Research Group, Charles University, Prague, "SOFA component model," [letztes Abrufdatum 18.03.2006]. [Online]. Available: http://nenya.ms.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html

[28] Eclipse, "EMFT – Eclipse Modeling Framework Technologie," [letztes Abrufdatum 10.03.2006]. [Online]. Available: http://www.eclipse.org/emft/projects/

[29] K. Ehrig, C. Ermel, and G. Taentzer, "Erstellung eines grafischen Editor-Plug-Ins mit Eclipse EMF und GEF," TU Berlin, Tech. Rep., 2005, [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://www.omondo.de/pdfs/ehrig_ermel_OS_02_05.pdf

[30] F. Eller and M. Kofler, *Visual C#: Grundlagen, Programmiertechniken, Windowsprogrammierung.* Addison-Wesley, 2005.

[31] EMF, "Generating an EMF Model," Juni 2005, [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://www.eclipse.org/emf/docs.php?doc=tutorials/clibmod/clibmod.html

[32] T. Eymann, *Digitale Geschäftsagenten: Softwareagenten Im Einsatz.* Berlin: Springer, 2003.

[33] P. František, B. Dušan, and R. Janecek, "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating," in *Proceedings of ICCDS 98, Annapolis, Maryland, USA.* IEEE CS Press, May 4-6 1998.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Entwurfsmuster.* München: Addison-Wesley, 2004.

[35] GMF, "GMF Tutorial Part 2," [letztes Abrufdatum 18.04.2005]. [Online]. Available: http://wiki.eclipse.org/index.php/GMF_Tutorial_Part_2#Another_Connection

[36] V. Grassi, R. Mirandola, and A. Sabetta, "From design to analysis models: a kernel language for performance and reliability analysis of component-based systems," pp. 25–36, 2005.

[37] J. P. Hamilton, *Object-Oriented Programming with Visual Basic .NET*, 1st ed., J. Petrusha, Ed. O'Reilly Media, 2002.

[38] IBM Corporation, “IBM Rational Software Architect,” [letztes Abrufdatum 26.02.2006]. [Online]. Available: http://www-128.ibm.com/developerworks/rational/products/rsa/

[39] M. Jeckle, *UML 2 glasklar*. Hanser, 2004.

[40] B. Jorn, “Model driven software development: An emerging paradigm for industrialized software asset development,” June 2004, [letztes Abrufdatum 28.03.2006]. [Online]. Available: http://www.softmetaware.com/whitepapers.html

[41] N. Kawane, “EPTUD : An Eclipse Plugin For Testing UML Design Models,” Master’s thesis, Department of Computer Science, Colorado State University, 2005, [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://www.klasse.nl/eclipse-update-site

[42] khussey@ca.ibm.com, “Eclipse Bugzilla: Import of multiple UML2-Models into EMF Project fails,” [letztes Abrufdatum 16.02.2006]. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=127631

[43] H. Koziolek and J. Happe, “A Quality of Service Driven Development Process Model for Component-based Software Systems,” University of Oldenburg, Graduate School Trustsoft, Tech. Rep., 2006.

[44] K. Krogmann, “Proposal einer Diplomarbeit - Entwicklung und Transformation eines EMF-Modells des Palladio Komponenten-Meta-Modells,” http://www.kelsaka.de/, [letztes Abrufdatum: 26.02.2006].

[45] L. Kung-Kiu and Z. Wang, “A Survey of Software Component Models,” University of Manchester, School of Computer Science, Tech. Rep., April 2005.

[46] ——, “A taxonomy of software component models,” in *31st Euromicro Conference*. IEEE Computer Society Press, 2005, pp. 88–95.

[47] J. Löwy, *Programming .NET Components*, 1st ed., J. Osborn, Ed. O’Reilly, 2003.

[48] F. Ludwig and F. Salger, “Werkzeuge zur domänenspezifischen Modellierung,” *OBJECTspektrum*, vol. 3, pp. 16–20, 2006, [letztes Abrufdatum 16.04.2006]. [Online]. Available: http://www.sigs.de/publications/os/2006/03/ludwig_salger_OS_03_06.pdf

[49] B. Meyer, *Object Oriented Construction*, 2nd ed. Prentice Hall, 1998.

[50] Microsoft, “COM: Component Object Model Technologies,” [letztes Abrufdatum 28.03.2006]. [Online]. Available: http://www.microsoft.com/Com/

[51] Microsoft MSDN, “C#-Programmierreferenz – Tabelle für Gleitkommatypen,” [letztes Abrufdatum 16.01.2006]. [Online]. Available: http://msdn.microsoft.com/library/deu/default.asp?url=/library/DEU/csref/html/vcreffloatingpointtypes.asp

[52] ——, “C#-Programmierreferenz – Tabelle ganzzahliger Typen,” [letztes Abrufdatum 16.01.2006]. [Online]. Available: http://msdn.microsoft.com/library/deu/default.asp?url=/library/DEU/csref/html/vcrefintegraltypes.asp

[53] ——, “.NET Framework-Klassenbibliothek – String-Klasse,” [letztes Abrufdatum 16.01.2006]. [Online]. Available: http://msdn.microsoft.com/library/deu/default.asp?url=/library/DEU/cpref/html/frlrfSystemStringClassTopic.asp

[54] J. Miller and J. Mukerji, “MDA Guide Version 1.0.1,” Object Management Group, Tech. Rep., 2003, [letztes Abrufdatum: 26.02.2006]. [Online]. Available: http://www.omg.org/docs/omg/03-06-01.pdf

[55] M. A. Mnich, “RandomGUID-Klasse,” [letztes Abrufdatum 10.03.2006]. [Online]. Available: www.JavaExchange.com

[56] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden, *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, 1st ed., ser. Redbooks. IBM, Februar 2004.

[57] O. Nierstrasz, *Object-Oriented Software Composition.* Prentice Hall, 1995, ch. Regular Types for Active Objects, pp. 99 – 121.

[58] C. Nock, *Data Access Patterns.* Addison-Wesley Professional, 2004.

[59] Object Management Group, “CORBA Scripting Language, v1.1,” version 3.0.3 [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/03-02-01

[60] ——, “Model Driven Architecture,” [letztes Abrufdatum 15.12.2005]. [Online]. Available: http://www.omg.org/mda/

[61] ——, “UML 2.0 Infrastructure convenience document,” [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/cgi-bin/doc?ptc/04-10-14

[62] ——, “UML 2.0 Superstructure Specification,” [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/05-07-04

[63] ——, “MetaObjectFacility(MOF) Specification – Version 1.4,” April 2002, [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[64] ——, “Common Object Request Broker Architecture: Core Specification,” März 2004, [letztes Abrufdatum 29.04.2006]. [Online]. Available: http://www.omg.org/docs/formal/04-03-02.pdf

[65] ——, “Meta Object Facility (MOF) 2.0 Core Specification,” 2005, [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/docs/ptc/04-10-15.pdf

[66] ——, “MOF 2.0/XMI Mapping Specification, v2.1,” September 2005, [letztes Abrufdatum 07.03.2006]. [Online]. Available: http://www.omg.org/docs/formal/05-09-01.pdf

[67] ——, “OCL 2.0 Specification – Version 2.0,” Juni 2005, [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/docs/ptc/05-06-06.pdf

[68] ——, “Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification,” November 2006, [letztes Abrufdatum 25.02.2006]. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf

[69] B. Oesterreich, *Die UML 2.0 Kurzreferenz für die Praxis*, 3rd ed. Oldenbourg, März 2004.

[70] Palladio Research Group, “The Palladio Research Project,” [letztes Abrufdatum: 15.12.2005]. [Online]. Available: http://se.informatik.uni-oldenburg.de/research/projects/Palladio

[71] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[72] R. Soley and OMG Staff Strategy Group, “Model Driven Architecture,” Object Management Group, Tech. Rep., 2000, [letztes Abrufdatum: 26.02.2006] White Paper. [Online]. Available: ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf

[73] Sun, “Enterprise Java Beans 3.0 - Proposed Final Draft,” Dezember 2005, [letztes Abrufdatum 28.03.2006]. [Online]. Available: http://java.sun.com/products/ejb/docs.html

[74] University of Kent – Computing Laboratory, “Object Constraint Language Library,” [letztes Abrufdatum 09.03.2006]. [Online]. Available: http://www.cs.kent.ac.uk/projects/ocl/index.html

[75] H. Wada, S. Takada, J. Suzuki, and N. Doi, “A Model Transformation Framework for Domain Specific Languages: An Approach Using UML and Attribute-Oriented Programming,” University of Massachusetts and others, Tech. Rep., 2005, [letztes Abrufdatum: 27.04.2005]. [Online]. Available: http://www.cs.umb.edu/~jxs/pub/sci05-mturnpike.pdf

[76] M. Wahler, “Using OCL to interrogate your EMF model,” IBM, August 2004, [letztes Abrufdatum 09.03.2006]. [Online]. Available: http://www.zurich.ibm.com/~wah/doc/emf-ocl/

[77] Wikipedia, “Globally Unique Identifier,” [letztes Abrufdatum 16.01.2006]. [Online]. Available: http://en.wikipedia.org/wiki/Globally_Unique_Identifier

[78] ——, “Switch (Computertechnik),” [letztes Abrufdatum 28.03.2005]. [Online]. Available: http://de.wikipedia.org/wiki/Switch_%28Computertechnik%29

[79] H. Wimmel, “ Das Wortproblem für Petri-Netze,” Juli 2004, [letztes Abrufdatum 29.03.2006]. [Online]. Available: http://parsys.informatik.uni-oldenburg.de/thesis/th_das_wortp.html

A.7. Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Diplomarbeit selbstständig angefertigt habe und alle Teile, die wörtlich oder inhaltlich anderen Quellen entstammen, als solche kenntlich gemacht und in das Literaturverzeichnis aufgenommen habe. Diese Arbeit wurde weder in dieser noch einer ähnlichen Form einer anderen Prüfungsbehörde vorgelegt.

Karlsruhe, den 9. Mai 2006

Klaus Krogmann