Modellierung der Struktureigenschaften von Subsystemen bei architekturellen Entwurfsentscheidungen in komponentenbasierten Systemen

Bachelorarbeit von

Yves R. Schneider

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	M. Sc. Axel Busch

12. Januar 2016 – 11. Mai 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 11.05.2016

. . . . . .

(Yves R. Schneider)

Kurzfassung

In Softwaresystemen werden bei der (Weiter-)Entwicklung in der Regel bereits fertige (Sub-)Systeme als ein Teil der Gesamtlösung eingesetzt. Dies benötigt jedoch fundiertes Fachwissen darüber, wie ein solches Subsystem einer Entwurfsentscheidung in das eigene Softwaresystem zu integrieren ist. Möchte ein Entwickler eine bestimmte Entwurfsentscheidung in einem bereits bestehenden Softwaresystem nachträglich umsetzen, ist ihm meist nicht klar, wie das Subsystem dieser Entwurfsentscheidung in die bestehende Architektur integriert werden kann.

In dieser Bachelorarbeit wird ein formales Modell, welches die Eigenschaften einer Entwurfsentscheidung abbildet, konzipiert und evaluiert. Für diesen Zweck wird das Strukturmodell als eine Erweiterung für das Palladio Komponentenmodell vorgestellt, mit welcher ganze Umbau-Maßnahmen für die Integration der Komponenten einer Entwurfsentscheidung formalisiert werden. Diese Erweiterung umfasst Informationen, wie die Struktur einer Entwurfsentscheidung und deren Einfluss auf die Qualität der Architektur, und kann dafür genutzt werden, um mit ihr Entwurfsentscheidungen im Kontext von Palladio wiederverwendbar zu formalisieren.

Für die Evaluation des Strukturmodells wurden mehrere Fallstudien auf das Strukturmodell angewendet, um dessen Anwendbarkeit und Nutzen zu zeigen. Hierfür wurde die Struktur von verschiedenen Entwurfsentscheidungen und deren Auswirkungen auf eine bestehende Architektur mit Hilfe des Strukturmodells modelliert. Abschließend wurde eine solches Strukturmodell einer Entwurfsentscheidung in ein bestehendes Softwaresystem eingesetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Diese Bachelorarbeit ist wie folgt aufgebaut: In diesem ersten Kapitel der Einleitung werden der Beweggrund und das Ziel der Bachelorarbeit sowie verwandte Arbeiten erläutert. Das darauffolgende zweite Kapitel beschreibt die benötigten Grundlagen für das bearbeitete Themengebiet. Im dritten Kapitel werden der Lösungsansatz und das daraus resultierende Metamodell im Detail erklärt. Das vierte Kapitel beinhaltet die Evaluation dieses Metamodells durch eine dargestellte Fallstudie. Im fünften Kapitel werden nochmals das Ergebnis der Bachelorarbeit zusammengefasst und die Grenzen des Metamodells aufgezeigt.

1.1. Motivation

In Softwaresystemen werden bei der (Weiter-)Entwicklung in der Regel bereits fertige (Sub-)Systeme als Teil der eigenen Gesamtlösung eingesetzt. Dies benötigt jedoch fundiertes Fachwissen darüber, wie das Subsystem einer Entwurfsentscheidung in das eigene Softwaresystem zu integrieren ist. Möchte ein Entwickler eine bestimmte Entwurfsentscheidung in einem bereits bestehenden Softwaresystem nachträglich umsetzen, ist ihm meist nicht klar, wie das Subsystem dieser Entwurfsentscheidung in die bestehende Architektur integriert werden kann und wie sich diese auf die Qualitätseigenschaften dieses veränderten Softwaresystems auswirkt. Die Architektur beeinflusst das Softwaresystem wesentlich, nicht nur in seiner angebotenen Funktionalität, sondern auch in welcher Qualität diese Funktionalität letztendlich realisiert wird. So wirken sich Entwurfsentscheidungen im Normalfall auf die Architektur und so auch auf ihre Qualitätseigenschaften aus. Dies betrifft nicht nur Entwurfsentscheidungen, welche in der Entwurfsphase getroffen werden, sondern auch nachträgliche Änderungen der Architektur eines bestehenden Systems.

1.2. Ziel

Um überhaupt untersuchen zu können, wie sich eine Entwurfsentscheidung vollständig in eine bestehende Architektur integrieren lässt, wird zunächst ein formales Modell benötigt, welches die Struktureigenschaften einer Entwurfsentscheidung abbildet. Es soll versucht werden, die Integration einer Entwurfsentscheidung in eine (bestehende) Architektur zu kapseln. Der Entwurf und die Evaluation eines solchen formalen Modells einer Entwurfsentscheidung und seine Auswirkungen auf das Systemmodell sind Hauptbestandteil dieser Bachelorarbeit.

Einer der großen Nutzen, welcher aus einem solchen formalen Modell gewonnen werden kann, ist, dass Entwurfsentscheidungen schneller in (bestehende) Softwaresysteme integriert werden können. Entwurfsentscheidungen werden von Komponentenentwicklern

spezifiziert und können von Softwarearchitekten einfach wiederverwendet werden. Der Softwarearchitekt benötigt jedoch fundiertes Fachwissen über die Komponenten einer Entwurfsentscheidung, um diese den Anforderungen gemäß in sein System integrieren zu können. Durch das erstellte Modell sollen Softwarearchitekten mit weniger Fachwissen eine Entwurfsentscheidung einfacher in ihr (bestehendes) Softwaresysteme integrieren können.

1.3. Verwandte Arbeiten

Omar Alam, Jörg Kienzle und Gunter Mussbacher beschreiben in [2] Concern-Driven Software Development (CDD) als ein neues Paradigma der Softwareentwicklung. Hierbei schlägt CDD die Verwendung einer dreiteiligen Schnittstellenbeschreibung vor, um die wiederverwendbaren Elemente einer Entwurfsentscheidung zu beschreiben. Die *variation*-Schnittstelle definiert die Auswirkungen auf die Gesamtqualität des Softwaresystems. Die *customization*-Schnittstelle ermöglicht es, den gewählten Concern auf einen spezifischen Wiederverwendungskontext anzupassen. Die *usage*-Schnittstelle definiert, wie ein individuell angepasster Concern zuletzt verwendet wird. Wenn nun ein Concern wiederverwendet werden soll, so wird zuerst mit Hilfe der *variation*-Schnittstelle die Konfiguration gewählt, welche die gewünschten Auswirkungen auf die Gesamtqualität des Softwaresystems hat. Im Anschluss wird der ausgewählte Concern mit Hilfe der *variation*-Schnittstelle von Entwicklern an den Kontext der Anwendung angepasst. Mit Hilfe der *usage*-Schnittstelle kann zum Schluss die bereitgestellte Funktionalität des Concern verwendet werden.

Softwareproduktlinien-Engineering ist ein auf *Variability Modeling* basierender Ansatz um die Wiederverwendung von Software zu erleichtern. Jedoch kann Variability Modeling auch die Modellierung von topologischen Strukturen der Software erfordern. Hierfür werden in [6] Datenverarbeitungspipelines als eine Anwendung des topologischen Variability Modeling vorgestellt. In der Arbeit wird gezeigt, wie diese Fähigkeiten angewendet werden können, um die Einsatzmöglichkeiten von Big Data Verarbeitungspipelines modellieren und konfigurieren zu können.

Sebastian Lehrig definiert in [14] Architectural Templates (ATs) als eine domänenspezifische Sprache (DSL), um ganze Architekturstile von Komponentenmodellen zu formalisieren. Hierbei werden die ATs zusätzlich mit Annotationen zur Skalierbarkeit ergänzt, als Folge dessen werden modellgetriebene Qualitätsanalysen mit Palladio im Bereich Skalierbarkeit ermöglicht. Da sich die Arbeit auf Software-as-a-Service (SaaS) Anwendungen konzentriert, liegt der Fokus auf der Skalierbarkeit dieser Anwendungen und der daraus resultierenden Performance.

2. Grundlagen

Die folgenden Abschnitte des Kapitels sollen einen Überblick der zugrundeliegenden Technologien des Forschungsfeldes der Bachelorarbeit geben. Auch auf die für den Lösungsansatz und der Evaluation dieser Arbeit relevanten Grundlagen und Technologien wird eingegangen.

2.1. Komponentenbasierte Entwicklung

Durch die Wiederverwendung von bereits erprobter Software wird nicht nur die Entwicklung neuer Softwaresysteme beschleunigt, sondern auch die Qualität dieser neu entwickelten Systeme verbessert. Bei wiederverwendbaren Softwarekomponenten werden einzelne miteinander kombinierbare Softwarebausteine zusammen gekapselt, sodass diese Komponente leicht von Dritten verwendet werden kann, ohne die einzelnen Softwarebausteine verstehen zu müssen. Im Handbuch der Softwarearchitektur wird eine Komponente wie folgt definiert:

> Komponenten sind modulare Teile eines Systems, die ihren Inhalt und somit komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen.

Über diese klar definierten Schnittstellen kann eine einzelne Softwarekomponente Dienste anfordern oder auch selbst bereitstellen. Wenn die Schnittstelle für den angeforderten oder bereitgestellten Dienst kompatibel ist, kann diese Komponente durch eine andere Komponente ersetzt werden. Aus dem Verbund von Einzelkomponenten kann so wieder ein vollständiges neues Softwaresystem erstellt werden.

2.1.1. Komponentenhierarchie

Der Lebenszyklus einer Softwarekomponente lässt sich nach [3] in mehrere Abschnitte unterteilen. An erster Stelle erfolgt die Spezifikation der Komponente. Dazu gehören die zur Verfügung gestellten Schnittstellen der Komponente, jedoch nicht zwingend ihre erforderlichen Schnittstellen. Diese Spezifikation kann noch unvollständig sein. Wenn nur spezifiziert wird, welcher Dienst die Komponente zur Verfügung stellt, wird die Instanz dieser Komponente als ihr *Provided Type* bezeichnet.

Wenn zusätzlich noch alle erforderlichen Schnittstellen spezifiziert werden, wird die Instanz der Komponente als *Complete Type* bezeichnet. Im Complete Type werden alle angebotenen und alle benötigten Schnittstellen definiert, jedoch noch keine interne Spezifikation der Komponente. Zu einer Provided Type Instanz kann es mehrere unterschiedliche Complete Type Instanzen geben.

Ein weiterer Abschnitt im Lebenszyklus der Komponente ist die interne Spezifikation der Komponente, also ihre Implementierung. Man spricht von einer *Implementation Type* Komponenten Instanz, wenn die Komponente intern vollständig implementiert wurde. Die Complete Type Instanz einer Komponente ist eine Abstraktion von einer Anzahl von möglichen Implementation Type Instanzen der Komponente [23, S. 44].

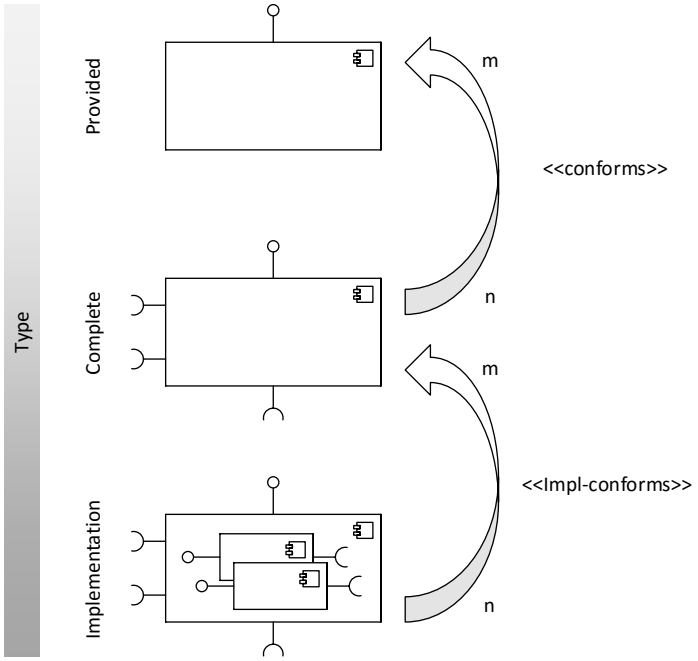


Abbildung 2.1.: Softwarekomponenten Typhierarchie nach [21]

2.2. Grundlagen der Modellierung

Um überhaupt etwas Komplexes wie eine komplette Softwarekomponente effektiv analysieren zu können, muss diese abstrahiert werden. Es wird eine stellvertretende Repräsentation der Softwarekomponente benötigt, welche sich auf bestimmte Aspekte beschränkt. Ungeachtet dieser Abstraktion soll diese Repräsentation dennoch Rückschlüsse auf eine reale Softwarekomponente ermöglichen. Im Handbuch der Softwarearchitektur [22, S. 94] wird ein Modell wie folgt definiert:

> Ein Modell beschreibt ein (reales) System in einer vereinfachten (abstrakten) Weise unter Verfolgung eines bestimmten Ziels.

Um solche Modelle von Softwarekomponenten analysieren zu können, müssen diese nach einheitlichen Regeln erstellt werden. Solche formale Regeln zum Erstellen von bestimmten Modellen werden als Metamodell bezeichnet. Im Handbuch der Softwarearchitektur [22, S. 94] wird ein Metamodell wie folgt definiert:

> Ein Metamodell ist eine präzise Festlegung der Bestandteile und Regeln zur Erstellung von Modellen. Es umfasst eine abstrakte Syntax, mindestens eine konkrete Syntax sowie statische und dynamische Semantik.

Durch ein Metamodell lässt sich so zum Beispiel eine spezifische Sprache formalisieren, welche eingesetzt werden kann, um Softwarekomponenten zu beschreiben. Da ein Metamodell selbst eine verkürzende Sicht der Definition einer Modellierungssprache ist, sind Metamodelle ebenfalls Modelle. Aus diesem Grund verfügen Metamodelle ebenfalls auch über ein Metamodell, das sogenannte Meta-Metamodell. Um nicht eine unendliche Reihe an Meta-Metamodellen zu benötigen, gibt es auch sich selbstbeschreibende Meta-Metamodelle. Im Handbuch der Softwarearchitektur [22, S. 96] wird die Eigenschaft eines sich selbstbeschreibenden Meta-Metamodells wie folgt definiert:

> Ein selbstbeschreibendes Meta-Metamodell ist dabei in der Lage, seine eigenen Bestandteile und Regeln mit seinen eigenen Mitteln zu definieren.

2.3. Palladio

Palladio ist ein Ansatz zur Simulation von komponentenbasierten Softwarearchitekturen. Palladio analysiert und bewertet hierbei die nichtfunktionalen Eigenschaften von Softwarearchitekturen wie Performance, Zuverlässigkeit, Wartbarkeit und Kosten. Palladio analysiert hierbei Softwaresysteme auf Modellebene, um so zum Beispiel Engpässe bei der Performance oder mögliche Probleme bei der Skalierbarkeit zu finden. Durch diese Simulation von Softwarearchitekturen werden „die Risiken und implizierten Kosten, die durch die Wahl der falschen Entwurfsentscheidung auftreten“, [7] reduziert. Durch diese Simulation von Softwarearchitekturen kann so beispielsweise die richtige Entwurfsentscheidung für eine Architektur gefunden werden.

Die Entwicklung des Palladio-Projektes begann bereits im Jahr 2003 an der Universität Oldenburg und wird heute am Karlsruher Institut für Technologie (KIT), Forschungszentrum Informatik (FZI) und der Technische Universität Chemnitz (TU Chemnitz) weiterentwickelt. Basierend auf der integrierten Entwicklungsumgebung Eclipse wurde eine integrierte Modellierungsumgebung für Palladio, die sogenannte Palladio-Bench, implementiert. Die Palladio-Bench Modellierungsumgebung ermöglicht Entwicklern, mit Hilfe des grafischen Editor Sirius Palladio Komponentenmodelle zu modellieren und diese anschließend zu analysieren [23].

2.3.1. Palladio Komponentenmodell

Das Palladio Komponentenmodell (Palladio Component Model, PCM) ist ein Metamodell, welches die formale Spezifikation von relevanten Informationen einer komponentenbasierten Architektur ermöglicht. Das Palladio Komponentenmodell wurde mit Fokus auf die Vorhersage von Quality-of-Service (QoS) Attributen wie Leistung und Zuverlässigkeit ausgelegt [12]. Das Metamodell des Palladio Komponentenmodells ist in Ecore definiert; Ecore ist eine weitere Meta-Metamodellierungssprache im Rahmen des Eclipse Modeling Framework (EMF).

2.3.2. PCM-Profil

In [13] wird von Kramer et al. eine zuverlässige Möglichkeit vorgestellt, das Palladio Komponentenmodell zu erweitern ohne das zugrunde liegende Metamodell verändern zu müssen. In ihrer Arbeit wurde Palladio um die Unterstützung von UML-Profilen und UML-Stereotypen erweitert. Mit Hilfe von UML-Profilen und Stereotypen lassen sich weitere Attribute zu vorhandenen Elementen des Palladio Komponentenmodells hinzufügen ohne das Metamodell abändern zu müssen.

2.3.3. PerOpteryx

PerOpteryx ist eine optionale Erweiterung für Palladio und standardmäßig in der PalladioBench enthalten. Das Ziel von PerOpteryx ist es, komponentenbasierte Softwarearchitekturen auf der Basis von modellbasierten Qualitätsvorhersagetechniken zu verbessern. Dazu nimmt PerOpteryx eine vollständige PCM-Instanz als Eingabe und erzeugt daraus neue PCM-Instanzen. PerOpteryx verwendet domänenspezifisches Wissen, um durch Manipulation bestimmter Parameter der ursprünglichen PCM-Instanz die Optimierung zu verbessern. Anschließend analysiert PerOpteryx die Qualitätsmerkmale der neuen PCM-Instanzen und versucht dabei, optimale Lösungen zu finden [16].

2.4. Softwarequalität

Bei dem Einkauf von Softwaresystemen sind nicht nur die bereitgestellten Funktionalitäten dieses Systems wichtig, sondern auch, in welcher Qualität diese Funktionalitäten des Systems bereitgestellt werden [22, S. 296]. Nach [22, S. 270] ist die Qualität eines Softwareprodukts die tatsächliche Ausprägung der Qualitätsmerkmale. Im folgenden Abschnitt sollen fünf ausgewählte Qualitätseigenschaften von Entwurfsentscheidungen beschrieben werden, deren Qualitätsmerkmale in [1] definiert und näher erörtert wurden. Hierbei wurden die Qualitätseigenschaften als die Ausprägung der Qualitätsmerkmale von Entwurfsentscheidungen beschrieben. Die ersten zwei Qualitätsmerkmale, Zuverlässigkeit und Wartbarkeit, können unter anderem von Palladio analysiert werden.

Zuverlässigkeit ist das Maß dafür, wie eine Entwurfsentscheidung die Fähigkeit eines Systems beeinflusst, sein Leistungsvermögen unter definierten Bedingungen über einen vorgegebenen Zeitraum aufrechtzuerhalten. Zum Beispiel würde die Entwurfsentscheidung, einen Load Balancer zur Lastverteilung einzusetzen, die Zuverlässigkeit eines Softwaresystems erhöhen.

Wartbarkeit ist das Maß, wie eine Entwurfsentscheidung die Fähigkeit eines Systems verändert, den Aufwand zu minimieren, welcher bei der Durchführung bestimmter Änderungen erforderlich ist. Beispielsweise würde die Entwurfsentscheidung, einen Dienst über einen Microservices bereitzustellen, die Wartbarkeit eines Softwaresystems erhöhen.

Interoperabilität ist das Maß dafür, wie eine Entwurfsentscheidung die Fähigkeit eines Systems zur möglichst nahtlosen Zusammenarbeit mit verschiedenen anderen Systemen verändert. Zum Beispiel würde die Entwurfsentscheidung, einen REST-Webservice zur Kommunikation mit anderen Systemen einzusetzen, die Interoperabilität eines Softwaresystems erhöhen.

Wiederherstellbarkeit ist das Maß, wie eine Entwurfsentscheidung die Fähigkeit eines Systems verändert, wenn durch den Wegfall von Komponenten betroffene Daten wiederherzustellen sind. Beispielsweise würde die Entwurfsentscheidung, einen Backup-Server einzusetzen, die Wiederherstellbarkeit eines Softwaresystems erhöhen.

Informationssicherheit ist das Maß dafür, wie eine Entwurfsentscheidung die Fähigkeit eines Systems, einen unberechtigten Zugriff auf Daten zu verhindern, verändert. Zum Beispiel würde die Entwurfsentscheidung, ein Intrusion Detection System einzusetzen, die Informationssicherheit eines Softwaresystems erhöhen.

2.5. Intrusion Detection System

Ein Intrusion Detection System (IDS) ist ein Softwaresystem, welches das Ziel hat, durch aktive Überwachung eines anderen Softwaresystems einen Angriff oder Missbrauch als solchen zu erkennen [17, 25]. Um überhaupt Ereignisse erfassen und sie danach analysieren und nach bestimmten Kriterien bewerten zu können, bedarf es mehrerer unterschiedlicher Softwarekomponenten. Als Intrusion Detection System wird eine Zusammenstellung von Softwarekomponenten bezeichnet, welche den gesamten Prozess von der Ereigniserfassung über die Analyse bis hin zur Alarmierung unterstützen. Nach [17, 25, S. 6] setzen sich komplette Intrusion Detection Systeme im allgemeinen aus folgenden Softwarekomponenten zusammen:

- Netz- / Hostsensoren
- Datenbankkomponenten
- Managementstation
- Auswertungsstation

2.6. Aspektorientierte Programmierung

Gregor Kiczales et al. stellen 1997 in [11] erstmals das Konzept der aspektorientierten Programmierung (AOP) vor. Die Mechanismen der aspektorientierten Programmierung ergänzen objektorientierte Sprachen um die Möglichkeit, eine oder mehrere Prozeduren durch mehrere Klassen hinweg miteinander zu verwenden. Die Idee dafür ist die Trennung der Komponenten der eigentlichen Anwendungslogik von den logischen Aspekten des Anwendungsprogramms. Nach Kiczales ist hierbei eine Komponente eine Eigenschaft des Softwaresystems, die in einer generalisierten Prozedur gekapselt werden kann. Im Gegensatz dazu ist ein Aspekt eine Eigenschaft, welche nicht in einer generalisierten Prozedur gekapselt werden kann [9]. Ein Join Point (dt. Verbindungspunkt) ist ein bestimmter Punkt im Programmfluss, an dem sich der Aspekt-Programmcode (advice) in den Programmfluss einhängen soll [24].

3. Ansatz

In dieser Bachelorarbeit wird ein formales Modell konzipiert, welches die strukturellen Eigenschaften einer Entwurfsentscheidung abbildet. Zu diesem Zweck wird ein Metamodell erstellt, mit welchem Informationen über die Struktur und die architekturelle Umsetzung abgebildet werden kann. Die Motivation, welche zur Modellierung und Speicherung einer Entwurfsentscheidung führt, ist oft das fachspezifische Wissen des Entwicklers über die entscheidenden (strukturellen) Eigenschaften der Entwurfsentscheidung und wie diese Entwurfsentscheidung in andere Systeme zu integrieren ist. Die Erfahrungen bei der Konzeption und Entwicklung einer solchen Entwurfsentscheidung werden mit Hilfe des Strukturmodells in eine einheitliche Hülle gefasst. Hierdurch soll vereinfacht werden, Entwurfsentscheidungen in bestehende Systeme zu integrieren. Dieses *Strukturmodell* können Komponentenentwickler nutzen, um damit Entwurfsentscheidungen zu formalisieren und diese in einem Repository bereitzustellen. Ein Vorteil der Modellierung von Entwurfsentscheidungen ist unter anderem, dass die resultierenden Modelle eine Dokumentations- und Kommunikationsgrundlage für Entwickler bilden. In dieser Arbeit wird darüber hinaus eine mögliche Grundlage für das automatische Einsetzen einer Entwurfsentscheidung für eine anschließende Analyse mit Palladio evaluiert. Um damit in einer späteren Entwicklungsphase (automatische) Qualitätsanalysen durchführen zu können, wird das Metamodell als Erweiterung des bestehenden Palladio Komponentenmodells konzipiert. Mit Palladio soll es dann möglich sein, die richtige Entwurfsentscheidung für eine bereits existierende Softwarearchitektur zu wählen. Anschließend soll Palladio die resultierende Qualität der Entwurfsentscheidung abhängig von der bereits existierenden Softwarearchitektur analysieren. Hierdurch soll Palladio die Risiken und implizierten Kosten reduzieren, welche durch die Wahl einer „schlechten“ Entwurfsentscheidung auftreten könnten.

Der Softwarearchitekt, welcher diese Entwurfsentscheidung in einem bereits bestehenden Softwaresystem nachträglich umsetzen will, soll so wenig Information wie möglich über die architekturelle Umsetzung einer solchen Entwurfsentscheidung benötigen. Aus diesem Grund muss der Entwickler, welcher eine Entwurfsentscheidung modelliert und in einem Repository ablegt, sich Gedanken machen, wie sein Subsystem der Entwurfsentscheidung später in jede bestehende Architektur integriert werden kann. Hierzu stellt das Strukturmodell Möglichkeiten bereit, um solche Informationen zu modellieren, welche in den Abschnitten 3.4 und 3.5 genau beschrieben werden.

Der sachliche und zeitliche Arbeitsablauf der Bachelorarbeit ist in Abbildung 3.1 dargestellt. Zunächst wird untersucht und bewertet, welche Arten von Entwurfsentscheidungen formal beschrieben werden können, sodass sich diese wiederverwendbar kapseln lassen. Das Ergebnis dieser Überlegungen wird im Abschnitt 3.2 beschrieben. Danach wird anhand einer konkreten Entwurfsentscheidung untersucht, welche architekturelle Struktur diese aufweist und wie sich diese formal beschreiben lässt. Im dritten Schritt wird

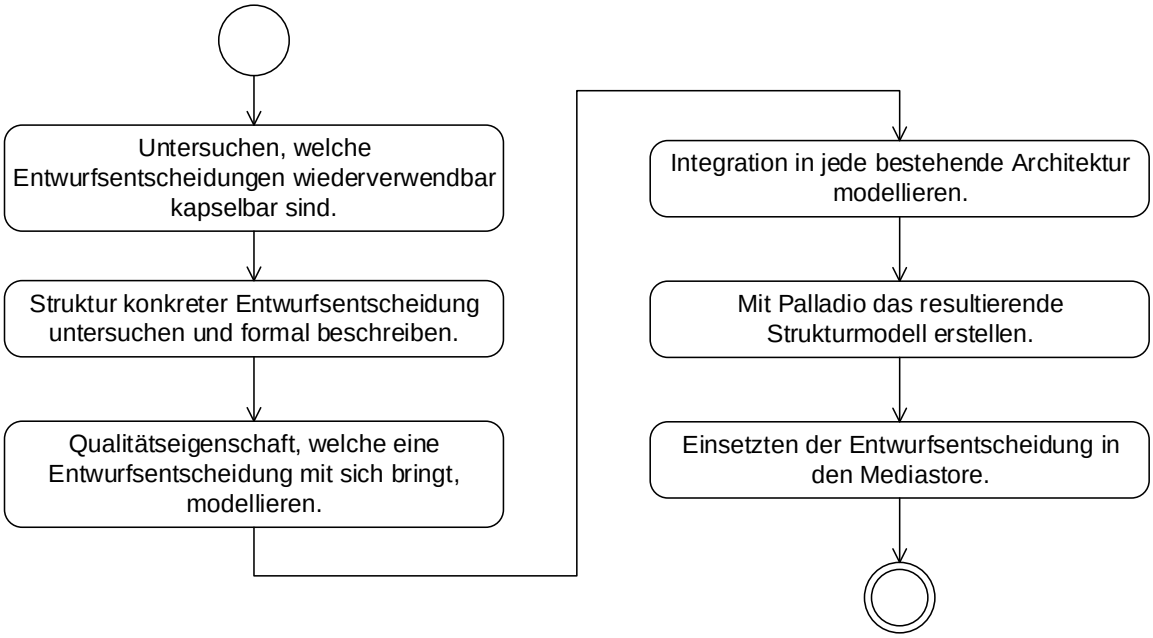


Abbildung 3.1.: Arbeitsablauf der Bachelorarbeit

durchdacht, wie sich die Qualitätseigenschaft, welche eine Entwurfsentscheidung mit sich bringt, modellieren lässt. Im nächsten Schritt wird überlegt, wie modelliert werden kann, wie sich eine Entwurfsentscheidung später in jede bestehende Architektur integrieren lässt. Hierzu gehören, wie und an welchen (Schnitt-)Stellen die Funktionalitäten der Entwurfsentscheidung integriert werden und auch an welchen Stellen die Komponenten der Entwurfsentscheidung in Betrieb gesetzt werden. Anhand dieser Untersuchungen wird das resultierende Strukturmodell erstellt, welches vollständig im Anhang auf Seite 49 abgebildet ist.

In den folgenden Abschnitten dieses Kapitels werden die einzelnen Elemente des Strukturmodells ausführlich erklärt. Es wird auch begründet und gegen andere Alternativen abgewogen, warum und wie etwas schließlich modelliert wird. Der Teil des *DesignDecisionRepository* wird in dem Abschnitt 3.2 erläutert. Die drei Elemente *DesignDecision*, *CompletedDesignDecision* und *DesignDecisionComponentImplementation* werden im Abschnitt 3.3 genauer behandelt. Der Bestandteil *Annotation* wird im Abschnitt 3.4.2 genauer beschrieben. Der Teil der *Transformation* wird im Abschnitt 3.5.4 und der Teil der *DeploymentMode* wird im Abschnitt 3.6 erläutert.

3.1. Subsysteme

Mit dem Strukturmodell lassen sich primär solche Entwurfsentscheidungen modellieren, deren Komponenten selbst wieder ein eigenes Softwaresystem bilden. Folglich bestehen diese Komponenten selbst wieder aus Teilsystemen. Dabei sind solche Subsysteme selbst noch keine vollständigen Anwendungen, sondern stellen nur Schnittstellen zur Verfügung, mit welchen Entwickler eine vollständige Anwendung implementieren können. Ein sol-

ches Subsystem kapselt demnach die wesentlichen Aspekte der Komponenten für einen Anwendungsbereich, welche auf die Bedürfnisse und Anforderungen einer konkreten Anwendung angepasst werden können [22, S. 319]. Solch ein Subsystem ist vergleichbar mit einem Framework auf Ebene von Komponenten. Nach [10] ist ein Framework eine „semi-vollständige Applikation, welche für andere Applikationen eine wiederverwendbare, gemeinsame Struktur zur Verfügung stellt“.

Um den Entwicklungsaufwand und die Entwicklungskosten gering zu halten, jedoch gleichzeitig die Zuverlässigkeit und Qualität von Softwarearchitekturen sicher zu stellen, werden in der Regel bereits fertige (Sub-)Systeme als Teil der eigenen Gesamtlösung eingesetzt. Dies beschränkt sich nicht nur auf die Entscheidung zur Entwurfszeit, wie zum Beispiel die Verwendung eines bestimmten Datenbankmanagementsystems. Sondern auch auf die Entscheidungen, welche erst später getroffen werden, um ein bereits laufendes System zu erweitern, wie zum Beispiel die Verwendung von Subsystemen wie ein Authentifizierungssystem oder ein Logging-System.

3.2. Repository für Entwurfsentscheidungen

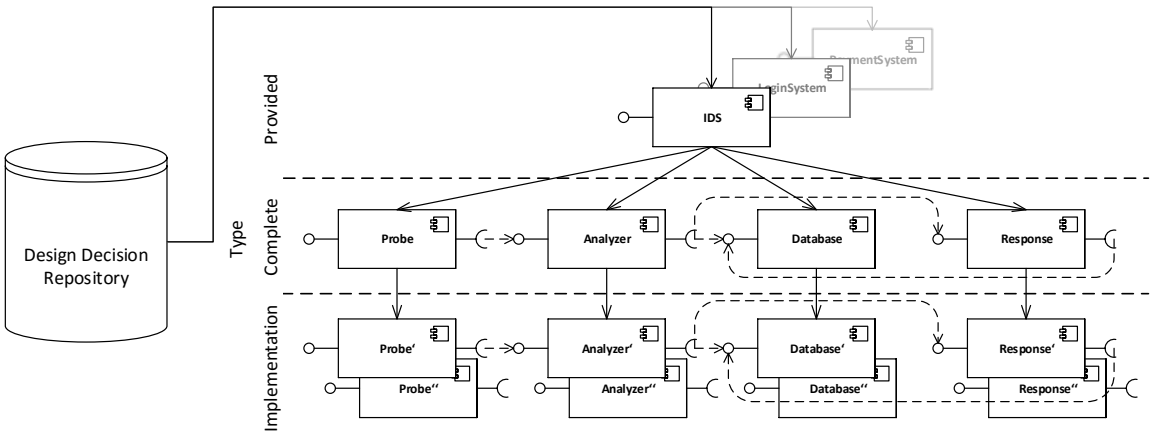


Abbildung 3.2.: Repository für Entwurfsentscheidungen mit Komponentenhierarchie

Das Repository für Entwurfsentscheidungen ist angelehnt an das Komponenten-Repository des Palladio Komponentenmodells [23, S. 136]. In Palladio lassen sich durch ein solches Repository bereits modellierte Komponenten einfach wiederverwenden. Architekten sollen komplette bereits modellierten Entwurfsentscheidungen durch ein Repository für Entwurfsentscheidungen einfach wiederverwenden können. Ein solches Repository ist beispielhaft als *Design Decision Repository* in Abbildung 3.2 schematisch dargestellt. Entwickler können die strukturellen Eigenschaften des Subsystems ihrer Entwurfsentscheidung abbilden und dieses Modell in einem Repository bereitstellen. Architekten können anschließend die Entwurfsentscheidungen aus einem Repository nutzen, um (bestehende) Systeme zu erweitern. Im folgenden Abschnitt 3.3 wird genau beschrieben, wie die strukturellen Eigenschaften einer Entwurfsentscheidung wiederverwendbar in eine einheitliche Hülle gefasst und in einem Repository gespeichert werden können.

3.3. Komponentenhierarchie

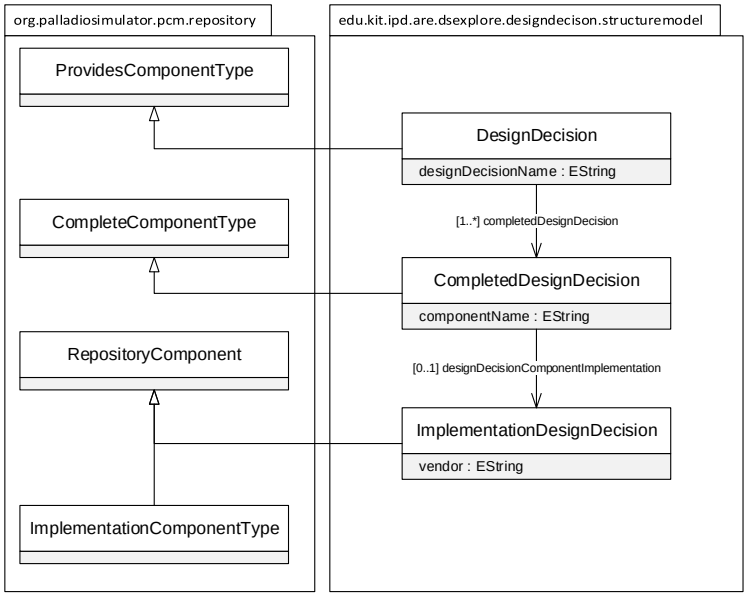


Abbildung 3.3.: Ausschnitt des Metamodells mit Typhierarchie von Komponenten

Die Herausforderung ist, Entwurfsentscheidungen im Kontext von Palladio so zu modellieren, dass sie für Architekten einfach auszuwählen sind, dass sie auch austauschbar gegen vergleichbare Entwurfsentscheidungen sind und dass modellierte Entwurfsentscheidungen wiederverwendbar gekapselt sind. Hierbei ist der Ansatz, eine Entwurfsentscheidung zusammenhängend sowohl als Provided, Complete und Implementation Type Komponenteninstanz zu modellieren. Jede Ebene der drei Komponenteninstanzen erfüllt eine der drei Anforderungen nach Einfachheit, Austauschbarkeit und Wiederverwendbarkeit. In Abbildung 3.3 ist die Komponentenhierarchie einer Entwurfsentscheidung schematisch dargestellt. In den drei folgenden Unterabschnitten werden die drei Komponentenhierarchieebenen ausführlich erklärt.

3.3.1. Provided Type Komponenteninstanz

Anfangs wird eine Entwurfsentscheidung als Provided Type Komponenteninstanz modelliert. Hierbei ist die Provided Type Instanz auf der höchst möglichen Abstraktionsebene, da sie die Qualtiätseinflüsse einer Entwurfsentscheidung lediglich definiert. Dies soll die Auswahl und Analyse der Entwurfsentscheidung nicht nur anhand der in Palladio bewährten Qualitätsdimensionen wie Performance, Zuverlässigkeit, Wartbarkeit und Kosten ermöglichen. Sondern auch weitere Kriterien der Softwarequalität, wie zum Beispiel Wiederherstellbarkeit, Interoperabilität oder Informationssicherheit umfassen. Durch die Reduzierung einer Entwurfsentscheidung auf ihre bereitgestellte Funktionalität kann diese anfangs abstrahiert werden, ohne über die benötigten Schnittstellen oder die interne Implementierung Aussagen treffen zu müssen. Diese Reduzierung soll es Entwicklern

in der ersten Phase des Architekturentwurfs leichter ermöglichen, zwischen verschiedenen Qualitätsattributen, welche das Softwaresystem später erfüllen soll, abzuwägen. Softwarearchitekten können so die Einflüsse auf die wichtigen Kriterien der zur Verfügung gestellten Softwarequalität einer Entwurfsentscheidung vorab grob bewerten und einschätzen. Soll beispielsweise eine Anforderung nach mehr Sicherheit umgesetzt werden, kann in einem Repository nach passenden Entwurfsentscheidungen gesucht werden, wie zum Beispiel ein Intrusion Detection System. Auch müssen Softwareentwickler oftmals zwischen verschiedenen Qualitätsattributen abwägen, welche ihre Softwaresysteme später erfüllen sollen. Soll beispielsweise das System sicherer werden, so kann dies unter Umständen höhere Kosten verursachen und womöglich das System auch etwas an Geschwindigkeit einbüßen. Aus diesem Grunde setzt sich eine Entwurfsentscheidung noch aus zwei weiteren Instanzen zusammen.

Im Kontext des Strukturmodells und Palladio wird die Provided Type Komponenteninstanz als *ProvidesComponentType* [23, S. 135] modelliert. Hierzu erhält das *ProvidesComponentType*-Element noch zusätzlich ein Attribut zum Benennen einer Entwurfsentscheidung, wie zum Beispiel *Authentifizierungssystem* oder *Logging-System*.

3.3.2. Complete Type Komponenteninstanz

Eine Ebene darunter befinden sich die Complete Type Komponenteninstanz. Im Complete Type der Entwurfsentscheidungen werden all die Komponenten definiert, welche benötigt werden, um die Entwurfsentscheidungen zu realisieren. Es kann sein, dass zur Realisierung einer Entwurfsentscheidung (Provided Type) mehrere Komponenten (Complete Type) benötigt werden. Der Complete Type abstrahiert die Komponenten einer Entwurfsentscheidung von deren internen Implementierungen und stellt einzig Schnittstellen dar, welche die Struktur und die Funktionalitäten der Komponenten spezifizieren. Im Complete Type werden bei allen Komponenten der Entwurfsentscheidung, sowohl die angebotenen als auch die benötigten Schnittstellenanschlüsse definiert. Dass bewusst keine Aussagen über die interne Implementierung dieser Komponenten getroffen werden, ermöglicht es, einfach aus verschiedenen Implementierungen eine Entwurfsentscheidung auszuwählen und auch auszutauschen. So gibt es beispielsweise Intrusion Detection Systeme von verschiedenen Herstellern, welche sich im allgemeinen aus den gleichen Arten von Komponenten zusammensetzen. Durch die Austauschbarkeit einer Entwurfsentscheidung durch verschiedene Implementierungen kann in der Folge mit Palladio (PerOpteryx) versucht werden, eine optimale Konfiguration zu finden. Indem Palladio alle Implementierungen nacheinander einsetzt und analysiert, kann mit Palladio direkt die richtige Implementierung (der richtige Hersteller) einer Entwurfsentscheidung für eine konkrete Softwarearchitektur ermittelt werden.

Im Kontext des Strukturmodells und Palladio wird die Complete Type Komponenteninstanz als *CompleteComponentType* [23, S. 125] modelliert. Hierzu erhält das *CompletedDesignDecision*-Element noch zusätzlich ein Attribut zum Benennen der Komponente einer Entwurfsentscheidung. Zum Beispiel stellt eine Entwurfsentscheidung eine *Network*- und eine *Storage*-Komponente zur Verfügung.

3.3.3. Implementation Type Komponenteninstanz

Die unterste Ebene sind die Implementation Type Komponenteninstanz. Sie definieren im Palladio Kontext eine vollständige Implementierungen aller von einer Entwurfsentscheidung benötigten Komponenten. Der Implementation Type bildet die logische Gruppierung von Komponenten, welche durch den Entwickler der Entwurfsentscheidung definiert wird und so durch einen Architekten wie gewohnt wiederverwendet werden kann. Im Kontext des Strukturmodells und Palladio werden die Implementation Type Komponenten Instanzen als alle Instanzen modelliert, welche das *RepositoryComponent*-Interface [23, S. 136] implementieren. Das *RepositoryComponent*-Interface ist der abstrakte Obertyp aller Komponenten, welche zur Wiederverwendung in Palladio in einem Komponenten-Repository gespeichert werden können. Dies können beispielsweise einfache Komponenten, zusammengesetzte Komponenten oder ganze Subsysteme sein. Jedes *ImplementationDesignDecision*-Element enthält noch zusätzlich ein Attribut zum Benennen des Herstellers der Entwurfsentscheidung. Zu einer Entwurfsentscheidung kann es unterschiedliche vollständige Implementierungen geben, zum Beispiel von unterschiedlichen Herstellern. Diese unterschiedlichen Implementierungen werden im Repository unter derselben Entwurfsentscheidung abgelegt. Des weiteren referenzieren die Elemente noch auf weitere Elemente, welche die Betriebsart und Integration der Komponenten der Entwurfsentscheidung bestimmen.

3.4. Analyse

Dieser Abschnitt behandelt die Frage, wie sich beim Modellieren verlässliche Aussagen über die Struktur eines beliebigen Softwaresystems treffen lassen, ohne dies vorab zu kennen. Das Ergebnis soll die Antwort sein, an welchen Stellen eine Komponente einer Entwurfsentscheidung in die bestehende Architektur eingefügt werden kann oder muss. Die Modellierung der Entwurfsentscheidung soll unabhängig von jedem späteren System sein, in welche sie eingesetzt wird. Folglich sollen bei der Modellierung so wenig wie möglich Annahmen über die Struktur eines Systems gemacht werden, in welches die Entwurfsentscheidung integriert wird. Die Frage nach der genauen technischen Umsetzung wird später im Abschnitt 3.5 erläutert. In diesem Abschnitt werden zwei unterschiedliche Methoden zur Analyse der Struktur einer Softwarearchitektur vorgestellt und erörtert.

3.4.1. Musterabgleich

Musterabgleich (engl. pattern matching) ist eine Verfahren, mit welchem anhand eines vorgegebenen Musters Strukturen von Softwaresystemen identifiziert werden sollen. Entwickler können so mit ihren Entwurfsentscheidungen auch noch bestimmte Muster abspeichern, womit zum Beispiel mögliche Integrationspunkte identifiziert werden könnten. In diesem Abschnitt werden zwei unterschiedliche Verfahren zum Musterabgleich erörtert, jedoch aufgrund der erörterten Probleme wurde das Musterabgleichsverfahren für nicht ausreichend eingestuft, um zuverlässig die Strukturen von Softwaresystemen zu identifizieren.

So kann zum Beispiel ein regulärer Ausdruck als ein solches Muster abgelegt werden. Mit Hilfe des regulären Ausdrucks ließe sich dann ein Musterabgleich auf den Bezeichnern von Komponenten, Schnittstellen oder Methoden realisieren. Eine Methode mit dem Bezeichner *login*, welche von einer Komponente mit dem Bezeichner *UserManager* bereitgestellt wird, ist mit hinreichender Wahrscheinlichkeit für die Benutzeranmeldung verantwortlich. Das Problem hierbei ist jedoch, dass es in der Regel keine festen Konventionen gibt, wie solche Bezeichner gewählt werden sollen, auch könnten solche Bezeichner mehrere kontextabhängige Bedeutungen haben. Zum Beispiel könnte der Bezeichner einer Datenbankkomponente *Database* oder einfach nur *DB* sein. Oder der Bezeichner könnte in einer anderen Sprache als die englische verfasst sein. Dies führt zu dem Schluss, dass ein regulärer Ausdruck zum Musterabgleich auf den Bezeichnern von Komponenten, Schnittstellen oder Methoden sich nicht eignet, um die Struktur einer Softwarearchitektur fehlerfrei zu analysieren.

Eine weitere Möglichkeit wäre ein Abgleich von eingesetzten Frameworks oder zumindest einzelne Module eines Frameworks. Eine Komponente, welche intensiv das Java-Framework *Hibernate* einsetzt, wird mit hinreichender Wahrscheinlichkeit mit der Datenhaltung zusammenhängen. Das Problem könnte hierbei sein, dass Frameworks, welche zu einen späteren Zeitpunkt entwickelt werden, nicht erkannt werden. Auch ist die Vielzahl an Frameworks, mit welchen sich ein Problem lösen lässt, selbst ein Problem. Der Entwickler der Entwurfsentscheidung müsste sich mit all diesen auseinandersetzen, um verlässliche Aussagen über die Struktur eines beliebigen Softwaresystems treffen zu können, um somit sicher identifizieren zu können, an welchen Stellen die Komponenten einer Entwurfsentscheidung in die bestehende Architektur eingefügt werden können. Dies führt zu dem Schluss, dass auf der Grundlage von verwendeten Frameworks die Struktur einer Softwarearchitektur sich nicht ohne großen Aufwand analysieren lässt.

3.4.2. Annotationen

Aufgrund der Probleme und Unzuverlässigkeiten, welche eine automatische Analyse der Architektur mit sich bringt, soll in diesem Abschnitt ein Verfahren gezeigt werden, mit welchem sich die Struktur von Softwaresystemen zuverlässig analysieren lässt. Auch wird gezeigt, wie sich dieses Verfahren modellieren lässt, sodass dieses mit einer Entwurfsentscheidung in einem Repository abgespeichert werden kann. Die zugrunde liegende Idee ist es Annotation zu definieren, welche von Entwicklern an relevante Stellen der Architektur eines Softwaresystems gesetzt werden müssen. Durch die Auszeichnung eines Softwaresystems mit zusätzlichen Informationen können so zum Beispiel mögliche Integrationspunkte identifiziert werden. Das Definieren dieser Annotation wird vom Entwickler der Entwurfsentscheidung vorgenommen. Dieser definiert alle wichtigen Integrationspunkte, welche benötigt werden, um seine Entwurfsentscheidung in ein anderes Softwaresystem zu integrieren. Das Setzen dieser Annotation an die entsprechenden Stellen eines Softwaresystems wird von dem Entwickler vorgenommen, welcher sein System um die Entwurfsentscheidung erweitern möchte, da dieser ohne größeren Aufwand verlässliche Aussagen über die Struktur des zu erweiternden Softwaresystems treffen kann.

Der Entwickler einer Entwurfsentscheidung definiert Annotation samt Beschreibung für die interessierte Funktionalität der Entwurfsentscheidung an bestehende Komponen-

ten, Schnittstellen oder Methoden in einem Softwaresystem. Der Entwickler, welcher Entwurfsentscheidungen umsetzen möchte, wird dann zum Beispiel mit Fragen über die bestehende Architektur angeleitet. Solche Fragen könnten zum Beispiel nach folgendem Schema aufgebaut sein: „Welche X ist/sind für Y verantwortlich?“ Wobei *X* entweder Komponente, Schnittstelle oder Methode sein kann und *Y* die interessierte Funktionalität der Entwurfsentscheidung an einer einzelnen Komponente, einer Schnittstelle oder einer Methode sein kann. Der Entwickler der Entwurfsentscheidung will zum Beispiel nicht, dass eine sicherheitskritische Komponente auf der selben Hardware zum Einsatz gebracht wird wie die Komponente für die grafische Benutzeroberfläche. Dann könnte zum Beispiel so eine Frage wie folgt lauten: „Welche Komponente ist für die grafische Benutzeroberfläche verantwortlich?“ In Abschnitt 3.6 wird gezeigt, wie modelliert werden kann, dass eine bestimmte Komponente der Entwurfsentscheidung nicht auf der gleichen Hardware zum Einsatz gebracht wird wie eine andere bestimmte Komponente aus dem bestehenden System. Eine weitere mögliche Frage könnte auch zum Beispiel lauten: „Welche Methoden sind für die Validierung von Eingangsdaten verantwortlich?“ In Abschnitt 3.5.3 wird gezeigt, wie modelliert werden kann, sodass ein Methodenaufruf um einen weiteren Methodenaufruf einer Komponente der Entwurfsentscheidung erweitert wird.

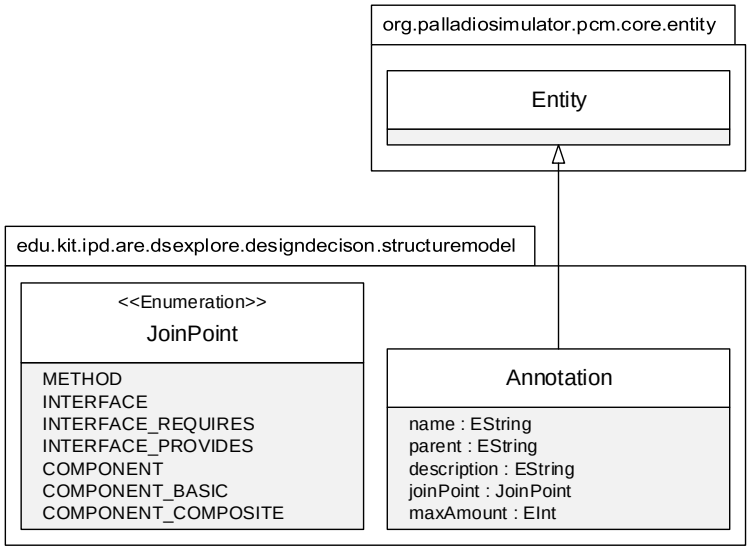


Abbildung 3.4.: Annotation Ausschnitt des Metamodells

In Abbildung 3.4 ist das UML-Diagramm des *Annotation*-Modellobjektes dargestellt. Ein jedes Annotation-Modellobjekt erbt von dem *Entity*-Modellobjekt [23, S. 107] des Palladio Komponentenmodell, hierdurch hat jede Annotation Schnittstellen für einen eindeutigen Identifikator und Namen. Die benötigten Attribute werden aus den entsprechenden Feldern *name* und *parent* erstellt. Die Namen von Annotationen sollten in der Regel Substantive oder Nominalphrasen sein. Der Name sollte immer so einfach und so beschreibend wie möglich sein, er sollte so gewählt werden, um für einen flüchtigen Betrachter sofort die eigentliche Absicht ihrer Nutzung aufzuzeigen. Falls sich der Name aus mehreren Wörtern zusammensetzt, wird immer der erste Buchstabe jedes Wortes großgeschrieben.

Der Identifikator setzt sich aus dem Namen der Entwurfsentscheidung und dem Namen der Annotation zusammen, zuerst kommt hierbei der Name der Entwurfsentscheidung gefolgt von dem der Annotation. Beide Namen werden mit einem *Punkt* (Satzzeichen) voneinander getrennt. Hierbei wird der Identifikator komplett in Kleinbuchstaben geschrieben. Zusätzlich hat jede Annotation noch ein Attribut *description : EString*, welches eine ausführliche textuelle Beschreibung der Annotation beinhaltet. Die Beschreibung wird vom Entwickler der Entwurfsentscheidung erstellt und dient dem Entwickler, welcher diese einsetzen möchte, als Hilfestellung dafür, wo genau er die Annotation setzen muss. Das Attribut *maxAmount : EInt* wird ebenfalls vom Entwickler der Entwurfsentscheidung definiert und bestimmt, wie oft eine Annotation maximal gesetzt werden darf. Ein Wert kleiner gleich Null bedeutet, dass der Entwickler, welcher die Entwurfsentscheidung in sein Softwaresystem einsetzen möchte, diese beliebig oft setzen darf. Das Attribut *joinPoint* mit dem zugehörigen Aufzählungstyp *JoinPoint* spezifiziert genauer den Punkt in der Architektur, an welchem die Annotation angefügt werden soll. Das Schlüsselwort *METHOD* verlangt, dass die Annotation an eine Methode angefügt wird. Das Schlüsselwort *INTERFACE* verlangt, dass die Annotation an einen beliebigen Schnittstellenanschluss angefügt wird. Das Schlüsselwort *INTERFACE_REQUIRES* verlangt explizit einen bereitgestellten Schnittstellenanschluss und das Schlüsselwort *INTERFACE_PROVIDES* verlangt explizit einen erforderlichen Schnittstellenanschluss. Das Schlüsselwort *COMPONENT* verlangt, dass die Annotation an eine beliebige Komponente angefügt wird. Das Schlüsselwort *COMPONENT_BASIC* verlangt explizit eine einfache Komponente und das Schlüsselwort *COMPONENT_COMPOSITE* verlangt explizit eine zusammengesetzte Komponente.

3.4.2.1. Annotation Profil

Die Fragestellung ist nun, wie solche zusätzlichen Annotationen mit Elementen des Palladio Komponentenmodells verknüpft und gespeichert werden können, ohne das entsprechenden Metamodell verändern zu müssen. Mit Hilfe von PCM-Profilen lassen sich Elemente des Palladio Komponentenmodells erweitern ohne das Metamodell abändern zu müssen. Alles was hierzu nötig ist, um Annotationen mit bestehenden Elementen zu verknüpfen und zu speichern, ist das Erstellen eines sogenannten MDSD-Profiles. In [15] wird ausführlich jeder Schritt erklärt, wie solche PCM-Profiles in Eclipse einzurichten sind, um anschließend ein neues Profil erstellen zu können, und auch, wie dann dieses Profil auf das Palladio Komponentenmodell angewandt wird. Hierdurch werden Annotationen als zusätzliche Informationen an Elemente des ursprünglichen Metamodells hinzugefügt.

In Abbildung 3.5 ist das UML-Diagramm des so erstellten Annotation-Profil dargestellt. Der Stereotyp *AnnotatedElement* erweitert die drei Modellobjekte *OperationSignature*, *OperationInterface* und *RepositoryComponent* des Palladio Komponentenmodells um eine Referenz auf das Modellobjekt *Annotation*, welche aus dem eigenen Metamodell für Entwurfsentscheidungen stammt. Das Modellobjekt *OperationSignature* entspricht im Palladio Kontext einer Operationsschnittstelle mit Parametern und Rückgabewert. Durch diese Erweiterung ist es möglich, Annotationen an Methoden anzuhängen. Das Modellobjekt *OperationInterface* entspricht im Palladio Kontext einer speziellen Art der Schnittstelle zum Aufruf von Operationen, hierzu referenziert das OperationInterface auf weitere OperationSignatures. Durch diese Erweiterung ist es möglich, Annotationen an Schnittstel-

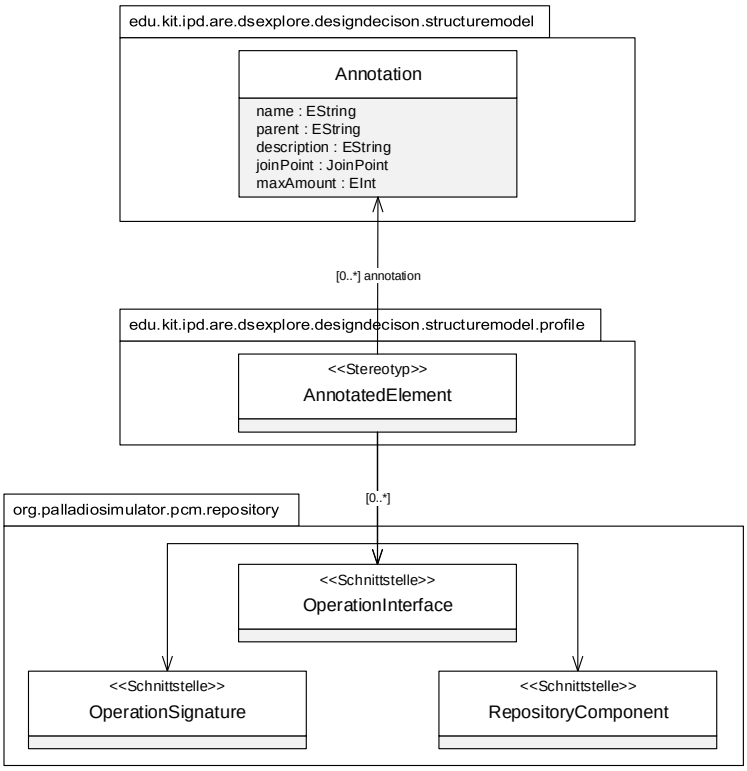


Abbildung 3.5.: Annotation Profil Ausschnitt des Metamodells

lenanschlüsse anzuhängen. Das Modellobjekt *RepositoryComponent* entspricht im Palladio Kontext der abstrakten Basisklasse aller Typen von Komponenten, welche Teil eines Komponenten-Repository sein können. Durch diese Erweiterung ist es möglich, Annotationen an Komponenten anzuhängen. Durch die Erweiterung von Modellobjekten des Palladio Komponentenmodells ist es zu einem möglich, Entwurfsentscheidungen um ihre bereitgestellten Annotationen zu erweitern und diese mit der Entwurfsentscheidung im Repository abzuspeichern. Ebenso ist es möglich, in das Modell eines bestehenden Systems an Methoden, Schnittstellen oder Komponenten Annotationen hinzuzufügen. Durch diese Erweiterung des Palladio Komponentenmodells soll eine spätere Integration in Palladio erfolgen, sodass sich die Auswirkungen der Entwurfsentscheidung anschließend analysieren lassen.

3.5. Integration

Dieser Abschnitt behandelt die Frage nach der technischen Umsetzung, wie das Subsystem einer Entwurfsentscheidung in eine bestehende Architektur integriert werden kann. Die Frage nach den Integrationspunkten, an welchen das Subsystem einer Entwurfsentscheidung in die bestehende Architektur eingefügt werden kann oder muss, wurde im vorherigen Abschnitt 3.4 erläutert. In diesem Abschnitt sollen nun die prinzipiellen Überlegungen des Ansatzes der technischen Integration beleuchtet werden.

Der erste Ansatz war es, sich zu überlegen, wie eine neue Beziehung zwischen Komponenten des bestehenden Softwaresystems mit den Komponenten der hinzuzufügenden Entwurfsentscheidung hergestellt werden kann. Der Lösungsansatz bei dieser Frage verfolgt die Idee, auf die etablierten Strukturmuster zurückzugreifen und zu versuchen, diese auf die Ebene von Softwarekomponenten zu übertragen. Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides beschreiben in ihrem 1994 herausgegebenen Buch *„Design Patterns. Elements of Reusable Object-Oriented Software“* Strukturmuster als Entwurfsmuster, welche sich damit beschäftigen, wie Klassen und Objekte zusammengesetzt sind, um größere Strukturen zu bilden [8, S. 155]. Hierbei verwenden die Strukturmuster auf Klassenebene Vererbung, um Schnittstellen oder Implementierungen zu komponieren. Im Buch werden insgesamt sieben verschiedene Strukturmuster vorgestellt: Adapter, Brücke, Kompositum, Dekorierer, Fassade, Fliegengewicht und Proxy.

Anstelle der Vererbung werden auf Ebene der Komponenten neue Komponenten hinzugefügt, welche diese Funktionalität übernehmen sollen. Diese Technik bringt jedoch einen Mehraufwand mit sich. Diese neuen Komponenten können unter Umständen höhere Kosten verursachen und womöglich kann das System durch diese auch etwas an Geschwindigkeit einbüßen. Das Verändern von Schnittstellen der Komponenten geht in der Regel mit einer Veränderung der internen Implementierung einer Komponente einher. Diese Änderung der internen Logik ist nicht ganz trivial und benötigt im Kontext von Palladio eine Modell-zu-Modell-Transformation. Von den sieben im Buch vorgestellten Strukturmustern erwies sich lediglich der Adapter als umsetzbar, ohne die bestehenden Komponenten grundlegend verändern zu müssen. Das Dekorierer-Strukturmuster erfordert eine Änderung am erforderlichen Schnittstellenanschluss des Dienstnehmers und das Proxy-Strukturmuster erfordert zusätzliche Logik in einer neu hinzugefügten Komponente. Die Einzelheiten dieser drei Strukturmuster werden in den folgenden drei Unterkapiteln genauer besprochen.

3.5.1. Proxy

Das Strukturmuster Proxy überträgt den Zugriff auf eine Komponente auf eine vorgelagerte Stellvertreterkomponente. So kann diese neue Proxy-Komponente den Zugriff der Dienstnehmer-Komponente auf die Dienstgeber-Komponente kontrollieren [8, S. 233].

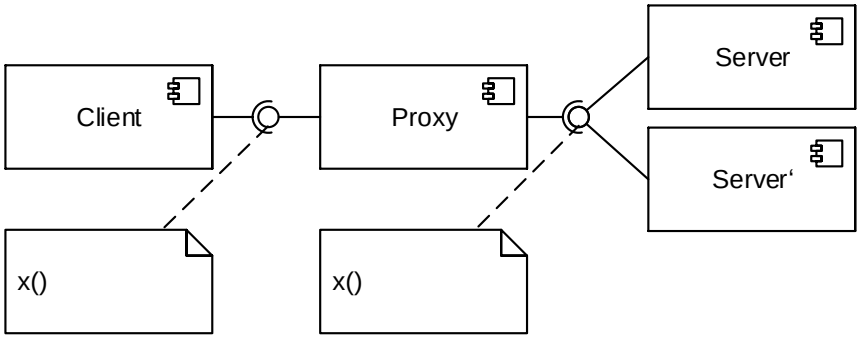


Abbildung 3.6.: Komponentendiagramm der Umsetzung des Proxy Entwurfsmusters

Voraussetzung hierfür ist jedoch, dass der neu hinzugefügte Dienstgeber Server' einen identischen bereitgestellten Schnittstellenanschluss wie der bereits bestehende Dienstgeber Server hat. Da der neue Dienstgeber von der ausgewählten Entwurfsentscheidung hinzugefügt wird, wird sich dieses Strukturmuster zur Integration der Entwurfsentscheidung in einem bestehenden System nicht immer und überall umsetzen lassen. Hinzu kommt, dass zusätzlich neben den Schnittstellenanschlüssen und der neuen Dienstgeber-Komponenten noch weitere Logik der Proxy-Komponente angehängt werden muss. Diese vervollständigende Logik dient der Regelung, an welche Komponente eingehende Anfragen weitergeleitet werden. Aufgrund dieser hohen Komplexität der Modellierung dieser Logik und der begrenzten Möglichkeiten der Integration wurde dieses Strukturmuster im Metamodell nicht umgesetzt. Mögliche Anwendungsszenarien, bei welchen es zu keinen Problemen wegen identischen bereitgestellten Schnittstellenanschlüssen kommen sollte, sind beispielsweise *Honeypot* oder ein *Load Balancer*. In der Informationssicherheit wird meist ein spezieller Server als ein Honeypot bezeichnet, welcher lediglich dazu dient, das Verhalten eines realen Servers oder sogar eins ganzen Rechnernetzes zu simulieren. So lassen sich in Rechnernetzen Honeypots einsetzen, um Informationen über Angriffsmuster und Angreiferverhalten zu erhalten [25, S. 12]. In der Informatik ist ein Load Balancer ein System, welches eine große Menge an Anfragen von Dienstnehmern (gleichmäßig) auf mehrere Dienstgeber verteilt. Sebastian Lehrig stellt in seiner Arbeit [14] ein Architectural Template für einen Load Balancer vor. Bei beiden Anwendungsszenarien ist jedoch vervollständigende Logik zur Regelung, an welche Komponente genau die eingehenden Anfragen weitergeleitet werden, vonnöten.

3.5.2. Dekorierer

Das Strukturmuster Dekorierer dient dazu, dynamisch ein Objekt um zusätzliche Funktionalitäten (Methoden) zu erweitern. So ist das Strukturmuster eine flexible Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionalitäten zu erweitern [8, S. 196].

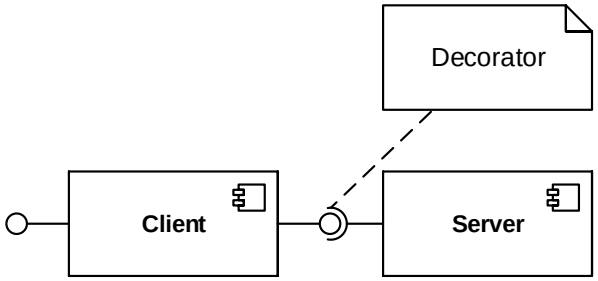


Abbildung 3.7.: Komponentendiagramm der Umsetzung des Dekorierer Entwurfsmusters

In diesem Ansatz wird die grundlegende Idee des Dekorierer-Strukturmusters auf Komponenten übertragen. Um einer bestehenden Komponente die Funktionalitäten einer neuen Komponente, welche durch eine hinzugefügte Entwurfsentscheidung hinzugekommen

ist, zur Verfügungen zu stellen, muss diese bestehende Komponente um einen neuen erforderlichen Schnittstellenanschluss erweitert werden. Hierbei werden jedoch der bestehenden Komponente nur neue Funktionalitäten (Methoden) zur Verfügung gestellt; die interne Integration dieser Funktionalitäten in der bestehenden Komponente wird nicht vorgenommen. Es ist vergleichsweise einfach zu modellieren, an welche Komponente neue Funktionalitäten (Methoden) einer Entwurfsentscheidung hinzugefügt werden sollen. Im Abschnitt 3.4 wurde ein Ansatz vorgestellt, um solche Komponenten in bestehenden Systemen zu identifizieren. Das Modellieren, auf welche Art und Weise diese neuen Funktionalitäten intern integriert werden, ist nicht möglich, da es möglicherweise vor dem Hinzufügen der Entwurfsentscheidung keine entsprechenden Strukturen in der bestehenden Komponente gab. Soll zum Beispiel eine bestehende Onlineanwendung um einen *User Manager* erweitert werden, kann die Funktionalität des neuen *User Manager* der *Session Manager* Komponente bereitgestellt werden. Wie aber diese Funktionalität in dem *Session Manager* in bestehende Strukturen implementiert wird, kann zuvor nicht modelliert werden.

3.5.3. Adapter

Das Strukturmuster Adapter stellt eine Art Hüllenklasse dar, welche die Schnittstelle einer Dienstgeber-Komponente in eine andere Schnittstelle überführt, welche von der Dienstnehmer-Komponente benötigt wird. Durch die Anwendung dieses Strukturmusters können Komponenten zusammenarbeiten, welche dies zuvor davor aufgrund von inkompatiblen Schnittstellen nicht konnten [8, S. 157].

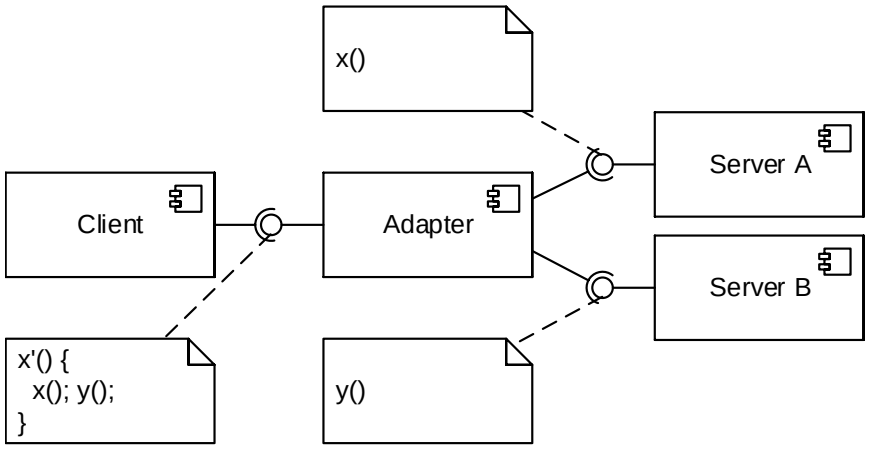


Abbildung 3.8.: Komponentendiagramm der Umsetzung des Adapter Entwurfsmusters

Damit das Einsetzen einer neuen Komponente ohne Änderung der bestehenden Komponenten funktioniert, wird sich an etablierten Techniken der aspektorientierten Programmierung orientiert. So wird die bereitgestellte Funktionalität einer Komponente der neu hinzugefügten Entwurfsentscheidung lediglich an bestehende Verbindungen mit angeheftet. Obwohl eine neue Adapter-Komponente zwischen zwei bestehenden Komponenten

eingesetzt wird, ändert sich so nichts an den Schnittstellenanschlüssen des zuvor bestehenden Dienstgebers und Dienstnehmers. Es werden lediglich zusätzliche Methoden einer neuen Komponente der Entwurfsentscheidung an bestehenden Methodenaufrufen mit angebunden. Diese Vorgehensweise ermöglicht es, mehrere Methoden durch mehrere unterschiedliche Komponenten hinweg miteinander zu verwenden. Die Idee dafür ist die Bestimmung von Verbindungspunkten in dem bestehenden System und die Bestimmung von Ausführungshinweisen in der Entwurfsentscheidung. Ein Verbindungspunkt (engl. join point) kann eine Methode, Schnittstelle oder eine vollständige Komponente sein und bildet einen bestimmten Punkt im bestehenden System, an dem sich die neue Funktionalität einer Entwurfsentscheidung einhängen soll. Ein Ausführungshinweis (engl. advice) definiert die Methode, welche ausgeführt wird, wenn ein zuvor definierter Verbindungspunkt erreicht wird. Die Modellierung im Metamodell einer solchen Adapter Transformation sowie Spezifikation von Verbindungspunkten und Ausführungshinweisen werden im folgenden Abschnitt 3.5.4 ausführlich besprochen.

3.5.4. Transformation

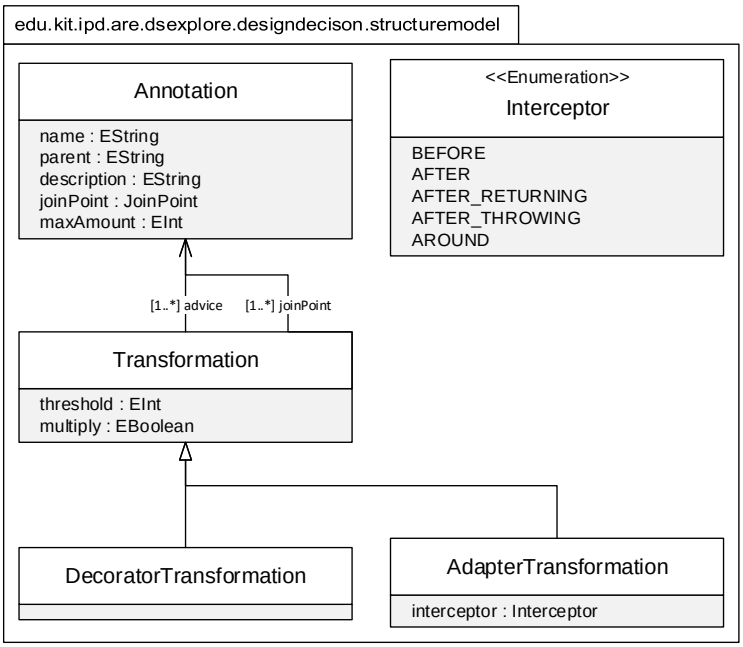


Abbildung 3.9.: Dekorierer- und Adapter-Transformation Ausschnitt des Metamodells

Durch die vom Architekten, welcher die Entwurfsentscheidung in sein System einsetzen möchte, gesetzten Annotationen in Verbindung mit den bereitgestellten Annotationen der Entwurfsentscheidung lässt sich modellieren, welche Methode oder Schnittstelle (Verbindungspunkt) um welche Methode (Ausführungshinweis) erweitert werden soll. Hierzu werden vom Entwickler beim Modellieren der Entwurfsentscheidung verschiedene Transformationsobjekte definiert. Hierbei beschreiben solche Transformationsobjekte des Strukturmodels keine Modell-zu-Modell-Transformationen aus der modellgetriebenen

Softwareentwicklung, sondern mit Hilfe der Transformationsobjekte sollen ausschließlich die grundlegenden Informationen modelliert werden, welche nötig sind um Modelltransformation zu beschreiben. Die Bestandteile eines solchen Objektes sind unter anderem zwei Listen (mehrere Referenzen auf Annotationen). Die erste Liste *joinPoint* sind die Annotationen, welche vom Benutzer gesetzt werden müssen, und so die Verbindungspunkte im bestehenden System referenzieren. Die zweite Liste *advice* sind eben die Annotationen, welche vom Entwickler der Entwurfsentscheidung gesetzt werden müssen, und so die Ausführungshinweise der Entwurfsentscheidung referenzieren. Durch das Attribut *multiply* wird modelliert, dass jedes Mal wenn eine Transformation ausgeführt wird, die Komponente, welche durch die *advice*-Annotation referenziert wird, auch neu mit in das System mit integriert wird. Hingegen, wenn das *multiply*-Attribut gleich *false* ist, wird die Komponente, welche durch die *advice*-Annotation referenziert wird, nur einmal in das System integriert, sodass alle weiteren Komponenten des bestehenden Systems lediglich auf die eine Komponente referieren.

3.5.4.1. Dekorierer

Bei der Dekorierer-Transformation werden lediglich die Schnittstellenanschlüsse oder Methoden, welche von der *advice*-Liste referenziert werden, an die Komponenten, welche in der *joinPoint*-Liste referenziert werden, als neue Funktionalitäten (Methoden) zur Verfügung gestellt. Die interne Integration der neuen Funktionalitäten in den bestehenden Komponenten muss anschließend von einem Entwickler vorgenommen werden.

3.5.4.2. Adapter

Bei der Adapter-Transformation werden die Methoden, welche von der *advice*-Liste referenziert werden, an bestehende Methoden, welche in der *joinPoint*-Liste referenziert werden, mit angeheftet. Die Variable *interceptor* mit dem zugehörigen Aufzählungstyp *Interceptor* spezifiziert genauer den Punkt im Programmfluss, an welchem der Ausführungshinweis ausgeführt werden soll. Das Schlüsselwort *BEFORE* sorgt dafür, dass der Ausführungshinweis vor der Methode des Verbindungspunktes aufgerufen wird. Das Schlüsselwort *AFTER* sorgt für die Ausführung des Ausführungshinweis nach der Ausführung der Methode des Verbindungspunktes, egal ob eine Exception geworfen wurde oder nicht. Bei dem Schlüsselwort *AFTER_RETURNING* wird der Ausführungshinweis nur dann mit ausgeführt, wenn die Methode des Verbindungspunktes regulär wieder verlassen wurde. Bei dem Schlüsselwort *AFTER_THROWING* wird der Ausführungshinweis nur dann mit ausgeführt, wenn die Methode des Verbindungspunktes eine Exception geworfen hat. Das Schlüsselwort *AROUND* sorgt dafür, dass der Ausführungshinweis einmal vor und auch einmal nach der Methode des Verbindungspunktes aufgerufen wird. Ein weiterer Bestandteil ist die Integer Variable *threshold*. Mit Hilfe dieser Variable lässt sich ein Schwellenwert definieren, ab welchem ein Adapter angewendet werden soll. Der Schwellenwert bezieht sich auf die Anzahl der Methoden eines Schnittellenanschlusses in dem bestehenden System. Wenn nur eine Methode in einem Schnittstellenanschluss als Verbindungspunkt identifiziert wurde, kann es sein, dass es sich aus Sicht der Performance

nicht lohnt, hier einen Adapter einzusetzen. Ist der Schwellenwert kleiner gleich Null, so wird immer ein Adapter eingesetzt.

3.6. Betriebsart

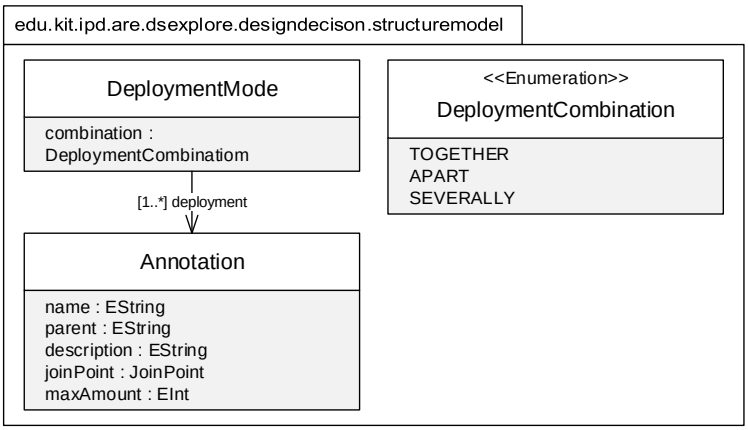


Abbildung 3.10.: Dekorierer- und Adapter-Transformation Ausschnitt des Metamodells

Das *DeploymentMode*-Element zusammen mit dem Aufzählungstyp *DeploymentCombination* im Metamodell modelliert den physischen Aspekt der Verteilung der Komponenten einer Entwurfsentscheidung. Dabei geht es darum, auf welcher Hardware-Ressource (Server) eine Komponente der Entwurfsentscheidung in Verbindung mit bestehenden Komponenten zum Einsatz gebracht wird. Hierbei geht es nicht um die Abbildung von Komponenten auf konkrete Ressourcen, sondern nur um das Verhältnis von neuen Komponenten, welche durch eine Entwurfsentscheidung hinzugefügt werden, und den bereits bestehenden Komponenten eines Systems. Jede Entwurfsentscheidung enthält Referenzen auf sogenannte DeploymentMode-Regeln, welche die physische Beziehung zwischen Komponenten regeln. Hierzu enthält das DeploymentMode-Element neben einem Attribut, welches den Betriebsmodus bestimmt, auch eine Liste mit Annotationen. Diese Annotationen können eine Mischung von bereitgestellten oder benötigten Annotationen der Entwurfsentscheidung sein. Der Aufzählungstyp *DeploymentCombination* stellt drei verschiedene Betriebsmodi zur Verfügung: *TOGETHER*, *APART* und *SEVERALLY*. *TOGETHER* verlangt, dass alle referenzierten Komponenten eines DeploymentMode-Elements zusammen auf einer Ressource zum Einsatz gebracht werden sollen. Hierüber kann zum Beispiel modelliert werden, dass eine bestimmte Komponente der Entwurfsentscheidung auf der selben Hardware wie eine andere Komponente des bereits bestehenden Systems in Betrieb genommen wird. Dies kann beispielsweise sinnvoll sein, wenn die Komponenten sehr viel miteinander kommunizieren und es zu so wenig wie möglich Mehraufwand durch die Netzwerkkommunikation kommen soll. *APART* verlangt, dass alle referenzierten Komponenten eines DeploymentMode-Elements getrennt voneinander auf unterschiedlichen Ressourcen zum Einsatz gebracht werden sollen. Dies kann beispielsweise sinnvoll sein, um sicherheitskritische Komponenten einer Entwurfsentscheidung von den Komponenten für

die Benutzeroberfläche fernzuhalten. *SEVERALLY* verlangt, dass alle referenzierten Komponenten eines DeploymentMode-Elements alleine auf eigenen Ressourcen zum Einsatz gebracht werden sollen.

4. Evaluation

Für die Evaluation des Strukturmodells, als eine Erweiterung des Palladio Komponentenmodell, werden mehrere Fallstudien durchgeführt. Hierbei werden die Fallstudien auf das im Ansatz konzipierte Strukturmodell angewendet, um dessen Anwendbarkeit und Nutzen zu zeigen. Das Ziel der Fallstudien ist es, zu untersuchen und bewerten, ob sich die strukturellen Eigenschaften einer Entwurfsentscheidung mit Hilfe des Strukturmodells abbilden lassen. Hierfür wird die Struktur von verschiedenen Entwurfsentscheidungen und deren Auswirkungen auf eine bestehende Architektur mit Hilfe des Strukturmodells modelliert. Für die abschließende Fallstudie wird das Strukturmodell anhand einer Entwurfsentscheidung evaluiert, ein Intrusion Detection System dem Media Store hinzuzufügen.

4.1. Bezahlsystem Entwurfsentscheidung

Die erste Entwurfsentscheidung, welche zur Evaluation des Strukturmodells herangezogen wird, ist die, ein Bezahlsystem in eine bestehende Anwendung, wie zum Beispiel einem Onlineshop, einzusetzen. Mit solch einem Bezahlsystem ist es möglich, den Benutzern einer Anwendung den Zugriff auf Daten, wie zum Beispiel Musik oder Filme, zu verwehren, wenn die Benutzer diese Daten nicht zuvor erworben haben.

Für die erste Fallstudie wird sich auf Complete Type der Entwurfsentscheidung fokussiert, um eine Anwendbarkeit auf wiederkehrenden Entwurfsentscheidungen zeigen zu können. Zu diesem Zweck, wird der Complete Type der Entwurfsentscheidung mit allen Komponenten modelliert, welche benötigt werden, um die Entwurfsentscheidungen zu realisieren, sowie die bereitgestellten und benötigten Annotationen mit den zugehörigen Transformationen und der Betriebsart. Für die Umsetzung eines Bezahlsystems sind folgende drei Komponenten notwendig: eine Paywall-, eine Payment- und eine Storage-Komponente.

4.1.1. Paywall-Komponente

Die Paywall-Komponente (Bezahlmauer) regelt den Zugriff auf die zahlungspflichtigen Daten. Die Komponente überprüft in einer Datenbank bei jedem Zugriff auf Daten, ob der entsprechende Benutzer bereits gezahlt hat oder nicht. Hat der Benutzer bereits gezahlt, leitet sie alle Anfragen einfach weiter. Ist der Benutzer aber nicht berechtigt, auf bestimmte Daten zuzugreifen, dann bricht die Paywall-Komponente die Transaktion ab, beispielsweise durch das Auslösen einer Exception. In Abbildung 4.1 ist das Modell der Paywall-Komponente mit bereitgestellten Annotationen dargestellt. Diese Komponente ist mit der Annotation *@Paywall* gekennzeichnet. Die Komponente stellt einen Schnittstellenanschluss mit einer Methode, welche für eine Benutzerkennung und eine Datenkennung

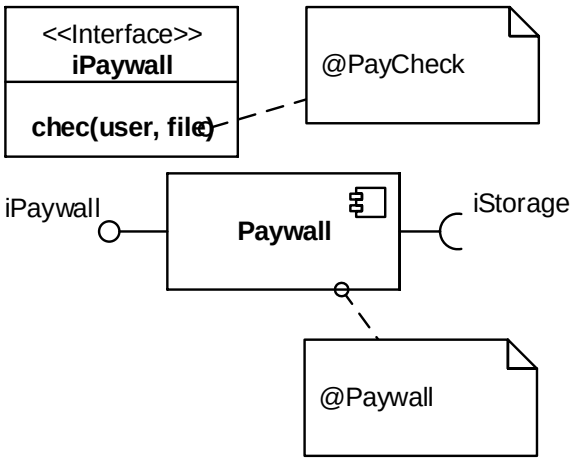


Abbildung 4.1.: Paywall-Komponente mit bereitgestellten Annotationen

eine Exception wirft, falls es sich um einen unerlaubtem Zugriff handelt. Diese Methode ist mit der Annotation *@PayCheck* versehen. Des weiteren benötigt die Paywall-Komponente noch einen weiteren Schnittstellenanschluss zu der Datenbankkomponente.

4.1.2. Payment-Komponente

Die Payment-Komponente (Bezahlsystem) ermöglicht es, Benutzern einer Anwendung den Zugriff auf Daten zu erwerben. Die Komponente speichert hierfür die Kontodaten eines Benutzers sowie die Datenkennung der gekauften Daten in der zugehörigen Datenbankkomponente. Auch ermöglicht die Komponente Geld Transaktionen über das Internet, indem sie sich mit einem Online-Bezahlsystem wie beispielsweise *PayPal* oder *Google Wallet* verbindet. Das Online-Bezahlsystem würde im Complete Type der Entwurfsentscheidung genau spezifiziert werden. In Abbildung 4.2 ist das Modell der Payment-Komponente mit bereitgestellten Annotationen dargestellt. Diese Komponente ist mit der Annotation *@Payment* gekennzeichnet. Die Komponente stellt einen Schnittstellenanschluss mit Methoden zur Verbindung mit einem Online-Bezahlsystem für den Erwerb von Daten bereit. Der Schnittstellenanschluss ist mit der Annotation *@Pay* versehen. Des weiteren benötigt die Payment-Komponente noch einen weiteren Schnittstellenanschluss zu der Datenbankkomponente.

4.1.3. Storage-Komponente

Die Storage-Komponente (Datenbank) speichert alle Transaktionen der Benutzer, welche durch die Payment-Komponente ausgelöst wurden, und erlaubt es der Paywall-Komponente zu überprüfen, ob ein Benutzer für bestimmte Daten bezahlt hat oder nicht. In Abbildung 4.3 ist das Modell der Storage-Komponente mit bereitgestellten Annotationen dargestellt. Diese Komponente ist mit der Annotation *@Storage* gekennzeichnet und stellt lediglich einen herkömmlichen SQL-Schnittstellenanschluss bereit.

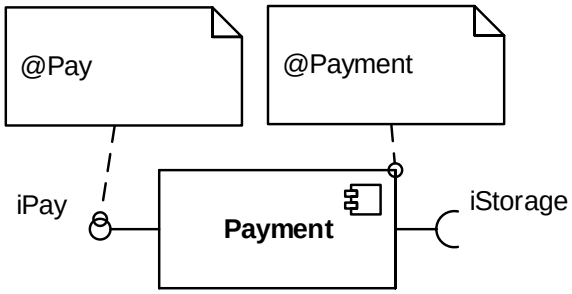


Abbildung 4.2.: Payment-Komponente mit bereitgestellten Annotationen

4.1.4. Annotation

Abbildung aller Annotationen, welche für die Integration und Betriebsart benötigt werden. Die Annotationen sind gegliedert nach den bereits zugewiesenen (bereitgestellt) und den Annotationen, welche noch an die Komponenten des bereits bestehenden Systems angefügt (benötigt) werden müssen. Die Felder *parent*, *name* und *description* wurden zur Übersicht nicht mit abgebildet. Das Feld *parent* entspräche immer dem Namen der Entwurfsentscheidung und das Feld *name* entspräche immer dem Namen der Annotation.

- Bereitgestellt:
  - @Pay
    - maxAmount = 1
    - joinPoint = METHOD
  - @PayCheck
    - maxAmount = 1
    - joinPoint = METHOD
  - @Payment
    - maxAmount = 1
    - joinPoint = COMPONENT
  - @Paywall
    - maxAmount = -1
    - joinPoint = COMPONENT
  - @Storage
    - maxAmount = 1
    - joinPoint = COMPONENT
- Benötigt:
  - @SafetyCritical
    - maxAmount = -1
    - joinPoint = COMPONENT
  - @DataAccess
    - maxAmount = -1
    - joinPoint = METHOD
  - @Database
    - maxAmount = -1
    - joinPoint = COMPONENT
  - @UserManager
    - maxAmount = 1
    - joinPoint = COMPONENT

4.1.5. Integration

Die zwei modellierten Transformationen genügen, um ein Bezahlsystem in ein bestehendes System zu integrieren. Die Adapter-Transformation stellt die Bezahlfunktionalität der

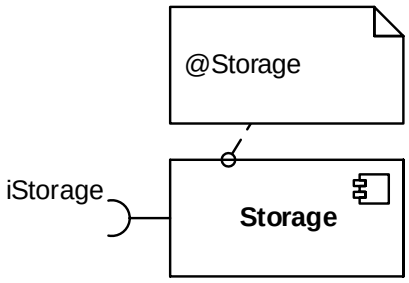


Abbildung 4.3.: Storage-Komponente mit bereitgestellten Annotationen

Benutzerverwaltung zur Verfügung, sodass die Implementierungen dieser Funktionalität einfach in der bestehenden Komponente umgesetzt werden kann. Die Decorator-Transformation erweitert alle bestehenden Methoden für den Datenzugriff um eine Bezahl-Kontrolle.

- AdapterTransformation
  - joinPoint = @DataAccess
  - advice = @PayCheck
  - threshold = -1
  - interceptor = BEFORE
- DecoratorTransformation
  - joinPoint = @UserManager
  - advice = @Pay
  - threshold = -1

4.1.6. Betriebsart

Die erste Regel des DeploymentMode modelliert, dass die Datenbank mit allen Zahlungsinformationen aus Sicherheitsgründen auf einer eigenen Hardware zum Einsatz gebracht wird. Die zweite Regel des DeploymentMode modelliert, dass die weiteren Komponenten nicht bei sicherheitskritischen Komponenten, welche möglicherweise kompromittiert werden können, zum Einsatz gebracht werden. Folglich modelliert diese Regel auch, dass die Payment-Komponente und Paywall-Komponente nicht auf der gleichen Hardware zum Einsatz gebracht werden.

- DeploymentMode
  - deployment = @Storage
  - combination = SEVERALLY
- DeploymentMode
  - deployment = [@SafetyCritical, @Payment, @Paywall]
  - combination = APART

4.2. Intrusion Detection System Entwurfsentscheidung

Diese zweite Fallstudie erweitert das Vorgehen der ersten Fallstudie um die Modellierung der Implementation Type Komponenten Instanzen. Diese Intrusion Detection System Fallstudie dient der grundlegend Fallstudie zur Evaluation des Strukturmodells. Zu diesem

Zweck wurde ein Intrusion Detection System umfassend untersucht und anschließend auch alle Komponenten in eine Anwendung eingebettet und mit dieser ausgeführt.

4.2.1. Allgemeine Struktur

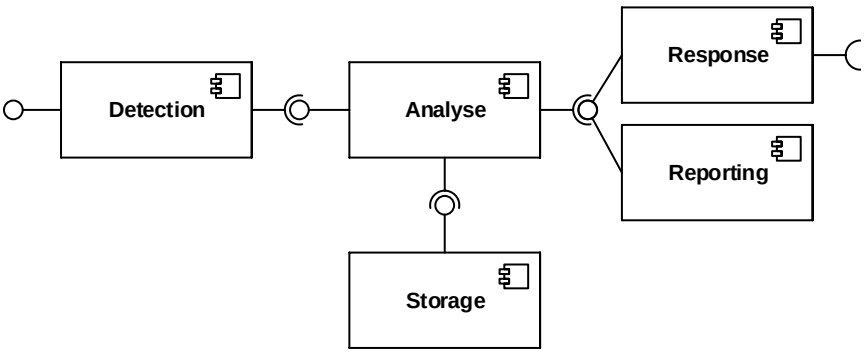


Abbildung 4.4.: Allgemeine Struktur der Komponenten eines IDS

Im Complete Type der Entwurfsentscheidung werden all die Komponenten definiert, welche benötigt werden, um die Entwurfsentscheidung zu realisieren. Ein Intrusion Detection System setzt sich im allgemeinen aus den gleichen Art von Komponenten zusammen [25, S. 6]: die Detection-Komponenten, eine Storage-Komponente, eine Analysis-Komponente, eine Reporting-Komponente und eine Response-Komponente. Die einzelnen Implementierungen der Komponenten werden in der Implementation Type Komponenten Instanz der zwei konkreten Implementierungen der Entwurfsentscheidung (AppSensor und OSSEC) genauer beschrieben. In Abbildung 4.4 ist die allgemeine Struktur der Komponenten eines Intrusion Detection System dargestellt.

4.2.1.1. Detection-Komponente

Damit Ereignisse aus einem Softwaresystem an ein Intrusion Detection System zum analysieren gesendet werden können, müssen die ursprünglichen Komponenten um zusätzliche Schnittstellen erweitert werden. Ein solcher Erfassungspunkt (detection point) kann an mehreren Stellen in ein bestehendes System hinzugefügt werden und kann verschiedenste Ereignisse (events) erkennen und an eine weitere Komponente des AppSensor zur Analyse senden. Solche Erfassungspunkte können verwendet werden, um böswillige Benutzer zu identifizieren, welche Schwachpunkte innerhalb einer Anwendung auskundschaften. Ein Beispiel hierfür wäre ein Benutzer, welcher ungewöhnlich viele Zeichen in das Feld für den Benutzernamen eingibt. Wenn Benutzernamen eine Grenze von 30 Zeichen haben und ein Benutzer gibt einen Benutzernamen mit 200 Zeichen ein, könnte dies ein Zeichen dafür sein, dass der Benutzer versucht, die Anwendung anzugreifen. In diesem Fall würde eine einfache Modifikation am bestehendem Programmcode ausreichen, um einen Erfassungspunkt einzufügen, der ein solches Ereignis meldet [19].

4.2.1.2. Analysis-Komponente

Die Analysis-Komponente ist der Kern eines Intrusion Detection System. Sie ist dafür verantwortlich, alle ihr gesendeten Ereignisse entgegen zu nehmen und im ersten Schritt in die Datenbank (Storage-Komponente) zu schreiben. Im zweiten Schritt analysiert die Komponente das Ereignis und überprüft, ob ein Angriff vorliegt. Falls ein Angriff erkannt wurde, wird dies ebenfalls in die Datenbank geschrieben. Darauf überprüft die Komponente, ob für einen solchen Angriff eine Gegenmaßnahme (response) definiert wurde und sendet diese gegebenenfalls an die Response-Komponente oder Reporting-Komponente.

4.2.1.3. Storage-Komponente

Die Storage-Komponente speichert alle Ereignisse (events), Angriffe (attacks) und Gegenmaßnahmen (responses). Die AppSensor-Referenzimplementierung umfasst derzeit verschiedene Implementierungen zur Datenspeicherung. Es werden neben SQL-Datenbanken und NO-SQL-Datenbanken auch eine In-Memory und dateibasierte Datenspeicherung unterstützt.

4.2.1.4. Response-Komponente

Die Response-Komponente ermöglicht es einer Anwendung, auf Angriffe, welche durch den AppSensor erkannt wurden, zu reagieren. Diese aktive Gegenmaßnahme sind ein wichtiger Teil des AppSensor-Konzepts. Eine solche Gegenmaßnahme kann zum Beispiel sein, einen Benutzer oder eine IP-Adresse für eine bestimmte Zeit zu sperren, sodass diese keine Dienste der Anwendung mehr in Anspruch nehmen können.

4.2.1.5. Reporting-Komponente

Die Reporting-Komponente ist dafür verantwortlich, Benachrichtigungen über Angriffsversuche an beispielsweise die Netzwerk-Administratoren zu senden.

4.2.2. Annotation

Abbildung aller Annotationen, welche für die Integration und Betriebsart der Intrusion Detection System Implementierungen benötigt werden. Die Annotationen sind gegliedert nach den bereits zugewiesenen (bereitgestellt) und den Annotationen, welche noch an die Komponenten des bereits bestehenden Systems angefügt (benötigt) werden müssen. Die Felder *parent* und *name* wurden zur Übersicht nicht dargestellt. Das Feld *parent* entspräche immer dem Namen der Entwurfsentscheidung und das Feld *name* entspräche immer dem Namen der Annotation. Die Bedeutung der einzelnen Annotationen, wird erst in der Implementation Type Komponenten Instanz der zwei konkreten Implementierungen der Entwurfsentscheidung (AppSensor und OSSEC) ausführlich erörtert.

Bereitgestellt

* @Analyse
* maxAmount = 1
* joinPoint = COMPONENT
* description = "Analysis-Komponente des IDS."

  - @DetectionEvent
    * maxAmount = 1
    * joinPoint = INTERFACE_PROVIDES
    * description = "Schnittstelle zum Senden von Ereignissen."
  - @DetectionException
    * maxAmount = 1
    * joinPoint = METHOD
    * description = "Methode zum Registrieren von Security-Exceptions."
  - @Reporting
    * maxAmount = 1
    * joinPoint = COMPONENT
    * description = "Komponente zur Berichterstattung."
  - @ResponseHandler
    * maxAmount = 1
    * joinPoint = COMPONENT
    * description = "Asynchrone Rückruffunktion für Gegenmaßnahmen."
  - @Storage
    * maxAmount = 1
    * joinPoint = COMPONENT
    * description = "Komponente zur Datenspeicherung."
- Benötigt
  - @SafetyCritical
    * maxAmount = -1
    * joinPoint = COMPONENT
    * description = "Komponente, die leicht durch Angreifer kompromittiert werden kann."
  - @SecurityEvent
    * maxAmount = -1
    * joinPoint = COMPONENT
    * description = "Komponente, die Ereignisse zur Analyse bereitstellt."
  - @SecurityException
    * maxAmount = -1
    * joinPoint = METHOD
    * description = "Methode, die durch eine SecurityException verlassen wird, wenn ein sicherheitskritisches Ereignis aufgetreten ist."
  - @UserManager
    * maxAmount = 1
    * joinPoint = COMPONENT
    * description = "Komponente zur Benutzerverwaltung."

4.2.3. AppSensor

AppSensor ist eines der Open-Source-Projekte des Open Web Application Security Project (OWASP). Das OWASP ist eine Non-Profit-Organisation, welche das Ziel hat, Anwendungen im Internet sicherer zu gestalten. Für dieses Ziel wurde unter anderem das AppSensor-Projekt ins Leben gerufen. Der AppSensor-Quelltext wird aktiv weiterentwickelt. Die aktuelle Version, mit welcher für die Fallstudie gearbeitet wird, ist die 2.2.0. Der AppSensor-Quelltext wird auf GitHub zur Verfügung gestellt und ist unter der MIT-Lizenz veröffentlicht. Mit Ausnahme der Verpflichtung zur Namensnennung des Urhebers, erlaubt die MIT-Lizenz, fast alles mit der Software zu machen [18]. AppSensor ist die Implementierung eines Intrusion Detection Systemen, welche als Referenz für andere Implementierungen von Intrusion Detection Systemen verwendet werden kann. Dabei definiert das AppSensor-Projekt nicht nur eine Referenzimplementierung, sondern auch ein komplettes Rahmenwerk

zur Methodik und Führung, wie Intrusion Detection Systeme entwickelt und bereitgestellt werden können [28].

Die AppSensor-Referenzimplementierung umfasst derzeit sechs verschiedene Ausführungsmodi, welche alle ihren eigenen Vor- und Nachteile für unterschiedliche Einsatzszenarien mit sich bringen. Für die Fallstudie wurde die Kommunikation der bestehenden Anwendung mit den Kernkomponenten des AppSensors mittels eines REST Webservice implementiert. Representational State Transfer (REST) ist ein Softwarearchitekturmodell für einen Webservice mittels des Hypertext Transfer Protocol (HTTP). Obwohl REST einfach, leichtgewichtig und effektiv ist, ermöglicht es doch eine hohe Flexibilität. REST wird auch in fast jedem Netzwerk unterstützt, was den Integrationsaufwand deutlich erleichtert. Der REST-Ausführungsmodus erlaubt auch ohne größere Probleme das Einbinden des AppSensor in Anwendungen, welche nicht mit Java implementiert sind. Auch können so mehrere (verteilte) Anwendungen überwacht werden.

In den folgenden Absätzen dieses Abschnittes sollen alle Komponenten der AppSensor-Implementierung erläutert werden. Mit besonderem Blick auf die Bestandteile, um welche eine bestehende Anwendung erweitert werden muss, damit der AppSensor als Intrusion Detection System in ein bestehendes System eingesetzt werden kann. Zu jeder Komponente gibt es zu Beginn einen Paragraphen, welcher die AppSensor-Implementierung genau beschreibt, und einen weiteren Paragraphen, welcher die resultierende Modellierung mit dem Strukturmodell beschreibt.

4.2.3.1. Detection-Komponente

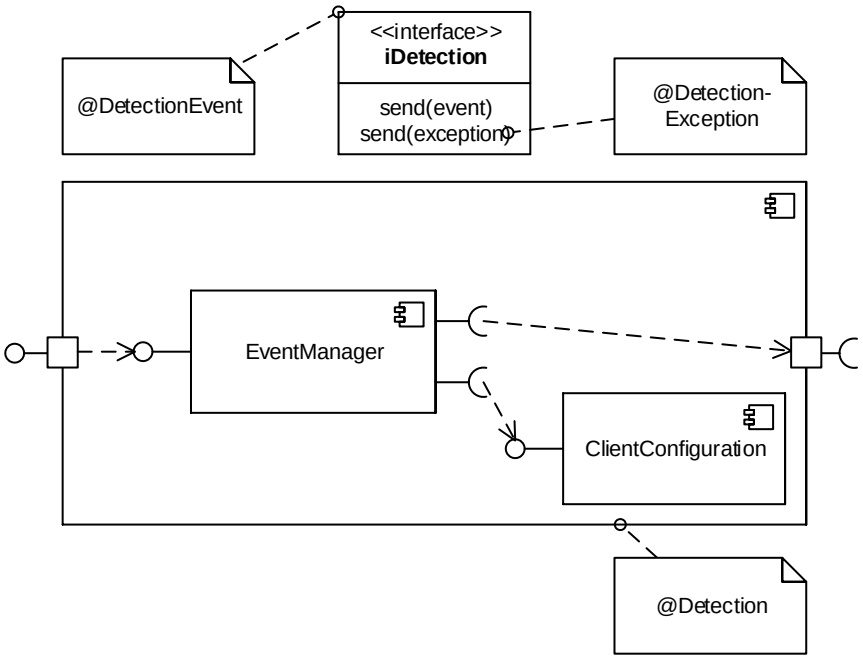


Abbildung 4.5.: AppSensor Detection-Komponente mit bereitgestellten Annotationen

In Abbildung 3.5 ist das Modell der Detection-Komponente dargestellt, welche sich aus einer *EventManager*-Komponente und einer *ClientConfiguration*-Komponente zusammensetzt. OWASP hat auf ihrer Webseite eine sehr umfangreiche Liste mit möglichen Erfassungspunkten veröffentlicht, an welcher Entwickler sich bei der Integration orientieren können [20]. Der bereitgestellte Schnittstellenanschluss der Detection-Komponente ist mit der Annotation *@Detection* gekennzeichnet. Die EventManager-Komponente stellt einen Schnittstellenanschluss zum Senden von Ereignissen an die Analysis-Komponente. Der bereitgestellte Schnittstellenanschluss der Detection-Komponente ist mit der Annotation *@DetectionEvent* versehen. Über diese Schnittstelle ist es anderen Komponenten möglich, Ereignisse an die Komponente des AppSensor zur Analyse zu senden. Die Schnittstelle stellt auch eine weitere Methode, welche mit der Annotation *@DetectionException* versehen ist, zum direkten Abfangen von SecurityExceptions bereit. Die EventManager-Komponente benötigt noch eine ClientConfiguration-Komponente, welche Informationen über die Analysis-Komponente bereithält wie Adresse und Zugangsdaten.

4.2.3.2. Analysis-Komponente

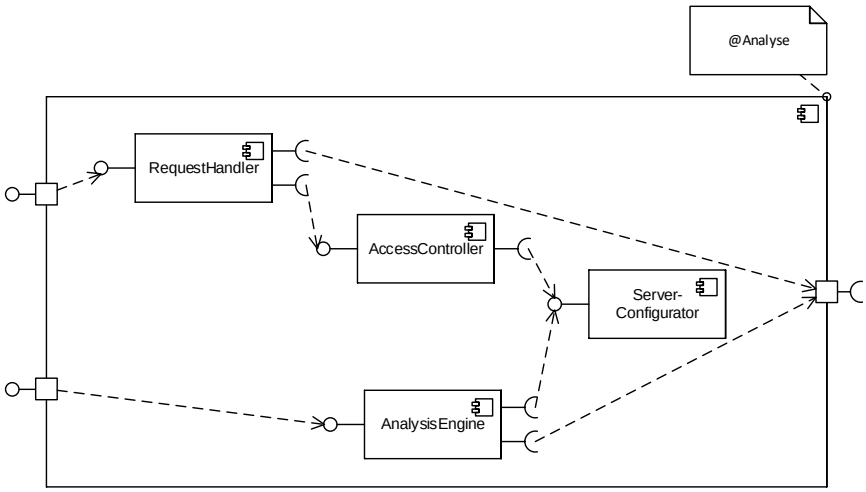


Abbildung 4.6.: AppSensor Analysis-Komponente mit bereitgestellten Annotationen

In Abbildung 4.6 ist das Modell der Analysis-Komponente dargestellt, welche sich aus einer *RequestHandler*-, einer *AccessController*-, einer *AnalysisEngine*- und einer *ServerConfigurator*-Komponente zusammensetzt. Diese Analysis-Komponente ist mit der Annotation *@Analyse* gekennzeichnet. Die RequestHandler-Komponente nimmt alle angehenden Ereignisse von Detection-Komponenten entgegen und sendet gegebenenfalls auch Anweisungen für Gegenmaßnahmen an eine Response-Komponente. Hierfür wird eine AccessController-Komponente benötigt, um zu überprüfen, ob eine Detection-Komponente berechtigt ist, Ereignisse zu senden. Die entsprechenden Rechte und an welche Response-Komponente die Anweisungen für Gegenmaßnahmen gesendet werden sollen wird mit der ServerConfigurator-Komponente geregelt. Auch ist mit dieser Komponente die Konfiguration für die AnalysisEngine-Komponente hinterlegt, wann für Ereignisse ein

Angriff vorliegt und wie die Gegenmaßnahmen für einen solchen Angriff aussehen. Die AnalysisEngine-Komponente ist der eigentliche Kern der Analysis-Komponente und des AppSensor, sie analysiert Ereignisse und stellt so fest, ob ein Angriff vorliegt und welche Gegenmaßnahmen hierfür eingeleitet werden sollen.

4.2.3.3. Storage-Komponente

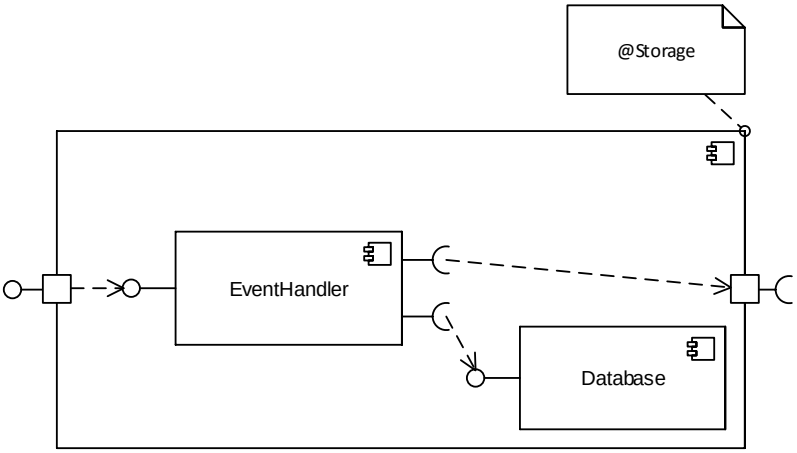


Abbildung 4.7.: AppSensor Storage-Komponente mit bereitgestellten Annotationen

In Abbildung 4.7 ist das Modell der Storage-Komponente dargestellt, welche sich aus einer *EventHandler*-Komponente und einer *Database*-Komponente zusammensetzt. Diese Storage-Komponente ist mit der Annotation *@Storage* gekennzeichnet. Die EventHandler-Komponente stellt einen Schnittstellenanschluss zum Empfangen von Ereignissen, Angriffen und Gegenmaßnahmen bereit. Auch kann sie der Analysis-Komponente auf Anfrage gespeicherte Ereignisse und Angriffe zur Analyse senden. Die eigentliche Speicherung der Daten wird von der Database-Komponente, einem Datenbankmanagementsystem, übernommen.

4.2.3.4. Response-Komponente

In Abbildung 4.8 ist das Modell der Response-Komponente dargestellt, welche sich aus einer *ResponseHandler*-, einer *EventManager*- und einer *ClientConfiguration*-Komponente zusammensetzt. Der bereitgestellte Schnittstellenanschluss der Response-Komponente ist mit der Annotation *@ResponseHandler* gekennzeichnet. Die ResponseHandler-Komponente prüft mit Hilfe des EventManagers, ob die Analysis-Komponente neue Angriffe erkannt hat und Gegenmaßnahmen bereitstellt. Die EventManager-Komponente benötigt einen Schnittstellenanschluss zum Empfangen der Gegenmaßnahme von der Analysis-Komponente. Hierfür benötigt der EventManager zusätzlich noch eine ClientConfiguration-Komponente, welche Informationen wie Adresse und Zugangsdaten für die Analysis-Komponente bereithält.

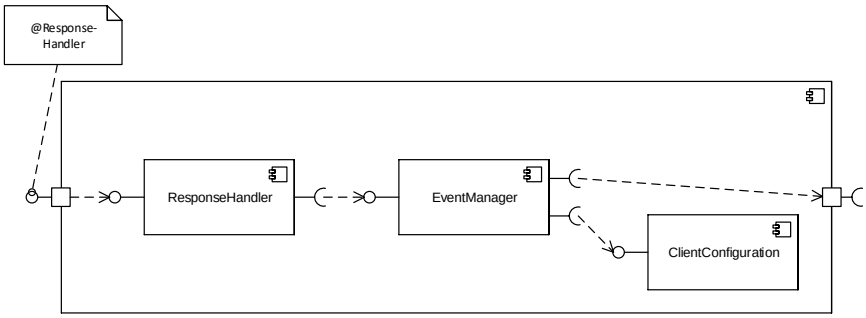


Abbildung 4.8.: AppSensor Storage-Komponente mit bereitgestellten Annotationen

4.2.3.5. Integration

Die bereitgestellten Funktionalitäten der Detection- und Response-Komponente müssen in das bestehende System integriert werden. Hierfür wurden drei verschieden Transformationen definiert. Die erste Transformation ist eine Adapter-Transformation, um SecurityException abzufangen, um aus diesen Ereignisse zur Analyse an den AppSensor zu senden. Bei dieser Transformation wäre es möglicherweise zweckmäßig, das *threshold*-Attribut auf einen höheren Wert zu setzen, da es sich aus Sicht der Performance nicht unbedingt lohnt, für eine einzige Methode einen neuen Adapter in das System einzusetzen.

Die zweite Transformation ist eine Decorator-Transformation und stellt Komponenten des bestehenden Systems die Funktionalität zur Verfügung, dass diese selbst Ereignisse erzeugen können und diese zur Analyse an den AppSensor senden können. Die Implementation dieser Funktionalität in einer bestehenden Komponente muss nach der Decorator-Transformation von einem Entwickler umgesetzt werden. Durch das Attribut *multiply* wird modelliert, dass jedes Mal, wenn eine solche Transformation ausgeführt wird, die Komponente, welche durch die *advice*-Annotation (in diesem Fall *@DetectionPoint*) referiert wird, auch erneut neu mit in das System integriert wird.

Die dritte Transformation ist eine Decorator-Transformation und stellt der Komponente des bestehenden Systems, welche für die Verwaltung von Benutzern zuständig ist, die Funktionalität zur Verfügung, dass diese bestimmte Benutzer für das System sperren können. Die Implementation dieser Funktionalität in der bereits bestehenden Komponente zur Benutzerverwaltung muss nach der Decorator-Transformation von einem Entwickler umgesetzt werden.

- AdapterTransformation
  - joinPoint = @SecurityException
  - advice = @DetectionException
  - threshold = -1
  - multiply = true
  - interceptor = AFTER_THROWING
- DecoratorTransformation
  - joinPoint = @SecurityEvent
  - advice = @DetectionEvent
  - threshold = -1
  - multiply = true
- DecoratorTransformation
  - joinPoint = @UserManager
  - advice = @ResponseHandler
  - threshold = -1
  - multiply = false

4.2.3.6. Betriebsart

Die ersten zwei Regeln des DeploymentMode modellieren, dass die Analyse- und die Storage-Komponente auf eigener Hardware zum Einsatz gebracht werden. Einmal aus Gründen der Performance, da diese zwei Komponenten den Kern des AppSensor bilden und die Analyse der Ereignisse rechenintensiv ist und auch mit vielen Datenbankanfragen verbunden. Der zweite Grund ist, dass diese zwei Komponenten nicht auf der gleichen Hardware mit anderen Komponenten befinden, welche möglicherweise von Angreifern kompromittiert werden könnten. Die nächsten drei Regeln des DeploymentMode modelliert, dass die Detection- und Response-Komponenten auf der gleichen Hardware wie die Komponenten des bestehenden Systems, welche die Funktionalitäten der Detection- oder Response-Komponenten benötigen.

- DeploymentMode
  - deployment = @Analyse
  - combination = SEVERALLY
- DeploymentMode
  - deployment = @Storage
  - combination = SEVERALLY
- DeploymentMode
  - deployment = [@UserManager, @ResponseHandler]
  - combination = TOGETHER
- DeploymentMode
  - deployment = [@SecurityEvent, @DetectionPoint]
  - combination = TOGETHER
- DeploymentMode
  - deployment = [@SecurityException, @DetectionPoint]
  - combination = TOGETHER

4.2.4. OSSEC

OSSEC ist ebenfalls ein quelloffenes und plattformübergreifendes Host-basiertes Intrusion Detection System. OSSEC kann unter anderem Log-Dateien analysieren, die Windows-Registry überwachen und Unix-basierte Rootkits erkennen. Des weiteren unterstützt OSSEC auch eine Alarmierung in Echtzeit und kann aktiv auf vordefinierte Ereigniss reagieren [4, S. 6]. Der Unterschied zu dem AppSensor Intrusion Detection System ist, dass OSSEC Log-Dateien als primäre Informationsquelle zu Angriffserkennung verwendet. Die verschiedenen OSSEC-Komponenten lassen sich auf den gängigen Betriebssystemen installieren, einschließlich Linux, OpenBSD, FreeBSD, MacOS, Solaris und Windows. Der OSSEC-Quelltext wird aktiv weiterentwickelt und ist auf GitHub zur Verfügung gestellt. OSSEC ist unter der zweiten Version der GNU General Public License (GNU GPLv2) veröffentlicht[27].

Die Implementierung eines Intrusion Detection Systems von OSSEC umfasst in der Regel drei verschiedene Komponenten, die Detection-Komponente, die Analysis-Komponente und die Reporting-Komponente. OSSEC ermöglicht es jedoch, auch eine aktive Response-Komponente zu integrieren.

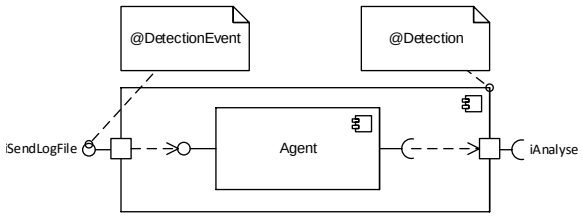


Abbildung 4.9.: Die Detection-Komponente sendet die Log-Dateien zum Analysieren

4.2.4.1. Detection-Komponente

In Abbildung 4.9 ist das Modell der Detection-Komponente dargestellt. Die Detection-Komponente ist dafür verantwortlich, Log-Dateien an die Analysis-Komponente weiterzuleiten. OSSEC stellt für diese Aufgabe bereits eine *Agent*-Komponente zur Verfügung. Diese Komponente stellt einen Schnittstellenanschluss mit Methoden bereit, welcher Log-Dateien entgegennimmt und diese zur Analyse weiterleitet. Dieser Schnittstellenanschluss ist mit der Annotation *@DetectionEvent* versehen. Des weiteren benötigt die Detection-Komponente noch einen weiteren Schnittstellenanschluss zur Analyse-Komponente.

4.2.4.2. Analysis-Komponente

In Abbildung 4.10 ist das Modell der Analyse-Komponente dargestellt. Diese Komponente ist mit der Annotation *@Analyse* gekennzeichnet. Die Analyse-Komponente ist dafür verantwortlich alle Ereignisse entgegenzunehmen, aufzubereiten und anschließend zu analysieren. Gegebenenfalls wird das Ergebnis der Analyse an die Reporting-Komponente weitergeleitet. Zum Empfangen der Log-Dateien, welche von der Agent-Komponente gesendet wurden, stellt OSSEC bereits eine *Remote*-Komponente bereit. Die Remote-Komponente leitet die empfangenen Log-Dateien fortlaufend an die *Logcollector*-Komponente weiter, welche die Dateien in ein für die Analyse einheitliches Format umwandelt. Danach werden die aufbereiteten Log-Dateien an die *Analysis*-Komponente zur eigentlichen Analyse weitergereicht. Um Administratoren über mögliche Angriffe informieren zu können, wird noch eine weiter Schnittstellenanschluss zur Reporting-Komponente benötigt.

4.2.4.3. Reporting-Komponente

In Abbildung 4.11 ist das Modell der Reporting-Komponente dargestellt. Die Reporting-Komponente ist dafür verantwortlich, Benachrichtigungen über Angriffsversuche an beispielsweise die Netzwerk-Administratoren zu senden. OSSEC stellt für diese Aufgabe bereits eine *Mail*-Komponente zur Verfügung. Diese Komponente stellt einen Schnittstellenanschluss mit Methoden bereit, welche Analyseergebnisse entgegennehmen, und diese per Mail weiterleiten.

4.2.4.4. Integration

Die Decorator-Transformation stellt die Funktionalität zum Senden von Log-Dateien allen Komponenten zur Verfügung, welche Log-Dateien erzeugen.

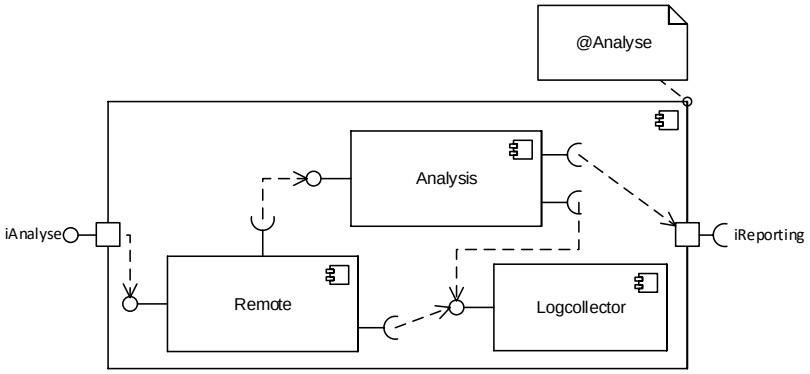


Abbildung 4.10.: OSSEC zusammengesetzte Analyse-Komponente

- DecoratorTransformation
  - joinPoint = @SecurityEvent
  - advice = @DetectionEvent
  - threshold = -1
  - multiply = true

4.2.4.5. Betriebsart

Die erste Regel des DeploymentMode modelliert, dass die Analyse-Komponente einmal aus Sicherheitsgründen und aus Gründen der Performance auf einer eigenen Hardware zum Einsatz gebracht wird. Die zweite Regel des DeploymentMode modelliert, dass die Reporting-Komponente nicht auf der gleichen Hardware wie möglicherweise kompromittierbare Komponenten zum Einsatz gebracht werden.

- DeploymentMode
  - deployment = @Analyse
  - combination = SEVERALLY
- DeploymentMode
  - deployment = [@SafetyCritical, @Reporting]
  - combination = APART

4.3. Media Store Fallstudie

Der *Media Store* ist ein in Java Enterprise Edition (JavaEE) implementiertes Softwaresystem, welches speziell für Fallstudien mit Palladio konzipiert wurde. Der Media Store ist eine webbasierte Anwendung, mit welcher mehrere Nutzer gleichzeitig mehrere verschiedene Audiodateien teilen und herunterladen können [26, S. 4]. Die Implementierung des Media Store wird aktiv weiterentwickelt und kann über ein (öffentliches) SVN-Repository heruntergeladen werden [5]. Die Implementierung des Media Store und der zugehörigen Palladio-Modelle ist unter der aktuellen Version der GNU General Public License (GNU GPLv3) veröffentlicht.

Die Architektur des Media Store, welche in Abbildung 4.12 dargestellt ist, ist eine komponentenbasierte Drei-Schichten-Architektur [26, S. 6]. Die erste Schicht (Präsentationsschicht) dient als Benutzungsschnittstelle, welche die Daten den Benutzern darstellt und

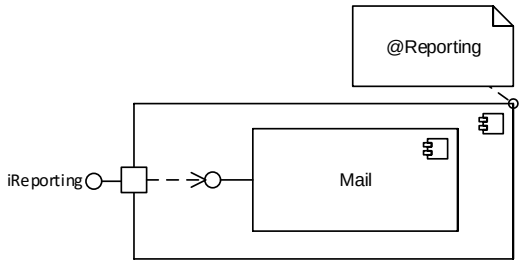


Abbildung 4.11.: Reporting-Komponente zur Benachrichtigung über Angriffsversuche

auch auf Benutzereingaben reagiert. Diese Schicht wird alleinig von der GUI-Komponente gebildet. Diese Komponente wurde mit der Annotation *@SecurityEvent* gekennzeichnet, um der GUI-Komponente die Funktionalität zur Verfügung zu stellen, dass diese selbst Ereignisse erzeugen und zur Analyse an die Analyse-Komponente senden kann. Die zweite Schicht (Logikschicht) beinhaltet die eigentliche Anwendungslogik des Media Store und beinhaltet unter anderem die UserManagement-, die MediaManagement- und die MediaAccess-Komponente. Die UserManagement-Komponente dient der Benutzerverwaltung und ermöglicht es Anwendern des Media Store, sich als Benutzer zu registrieren, anzumelden und auch wieder abzumelden. Diese Komponente ist ebenfalls mit der *@SecurityEvent*-Annotation und darüber hinaus auch mit der *@UserManager*-Annotation gekennzeichnet. Die MediaManagement-Komponente ist für das Hochladen und Herunterladen der Audiodateien verantwortlich und wurde ebenfalls mit der *@SecurityEvent*-Annotation gekennzeichnet. Die MediaAccess-Komponente ist für das direkte Abrufen der Audiodateien und deren Metadaten aus der Datenhaltungsschicht zuständig. Der Schnittstellenanschluss der MediaAccess-Komponente, welcher diese mit der Datenbank-Komponente für die Verwaltung der Benutzerdaten verbindet, ist mit der Annotation *@SecurityException* gekennzeichnet. Die dritte Schicht (Datenhaltungsschicht) ist für das Abspeichern und Laden aller Daten verantwortlich und beinhaltet zwei Datenbanken. Die DataStorage-Komponente ist für das Speichern aller Audiodateien zuständig und die DB-Komponente ist für das Speichern der Benutzerdaten zuständig.

4.3.1. AppSensor

Auf die drei Komponenten, welche mit der Annotation *@SecurityEvent* gekennzeichnet sind, kann anschließend die zuvor definierte Dekorierer-Transformation angewendet werden. Bei diesen Dekorierer-Transformationen werden der UserManagement-, der MediaManagement- und der GUI-Komponente neue Funktionalitäten (Methoden) der Detection-Komponente zur Verfügung gestellt. Die interne Integration der neuen Funktionalitäten, Ereignisse zur Analyse an weitere AppSensor-Komponenten senden zu können, kann anschließend einfach in den bestehenden Komponenten von einem Entwickler vorgenommen werden. Durch die Adapter-Transformation an dem Schnittstellenanschluss der MediaAccess-Komponente, welcher durch die Annotation *@SecurityException* gekennzeichnet ist, werden alle Methoden des Schnittstellenanschlusses um einen Methodenaufruf der Methode der Detection-Komponente, welche durch die Annotation *@DetectionException*

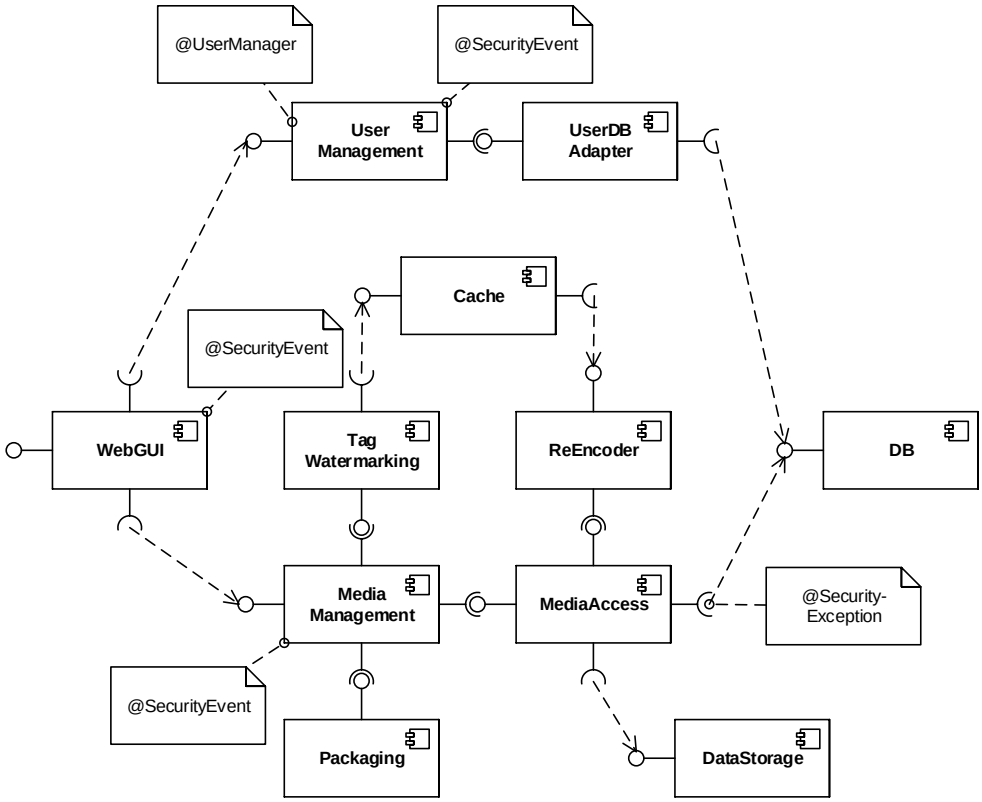


Abbildung 4.12.: Komponentendiagramm des mit Annotationen versehenden Media Store

gekennzeichnet ist, erweitert. Durch das Schlüsselwort *AFTER_THROWING* der Adapter-Transformation wird diese Methode nur dann mit ausgeführt, wenn die Methode des Verbindungspunktes eine Exception auslöst. Die letzte angewandte Transformation zur Integrierung des AppSensor ist eine Dekorierer-Transformation zum Einbinden der Response-Komponente. Hierfür wird die bestehende UserManagement-Komponente um einen Schnittstellenanschluss zu einer weiteren Detection-Komponente erweitert. Das resultierende Komponentendiagramm der Integration des AppSensor in den Media Store ist in in Abbildung 4.13 dargestellt.

4.3.2. OSSEC

Um nun OSSEC anstelle des AppSensor als Intrusion Detection System in den Media Store zu integrieren, reichen die zuvor gesetzten Annotationen ebenso aus. Auf die drei Komponenten, welche mit der Annotation *@SecurityEvent* gekennzeichnet sind, werden ebenfalls drei Dekorierer-Transformationen angewendet. Bei diesen Dekorierer-Transformationen werden der UserManagement-, der MediaManagement- der GUI-Komponente neue Funktionalitäten hinzugefügt, um Log-Dateien an die Analyse-Komponente senden zu können. Das daraus resultierende Komponentendiagramm der Integration des OSSEC Intrusion Detection System in den Media Store ist in in Abbildung 4.14 dargestellt.

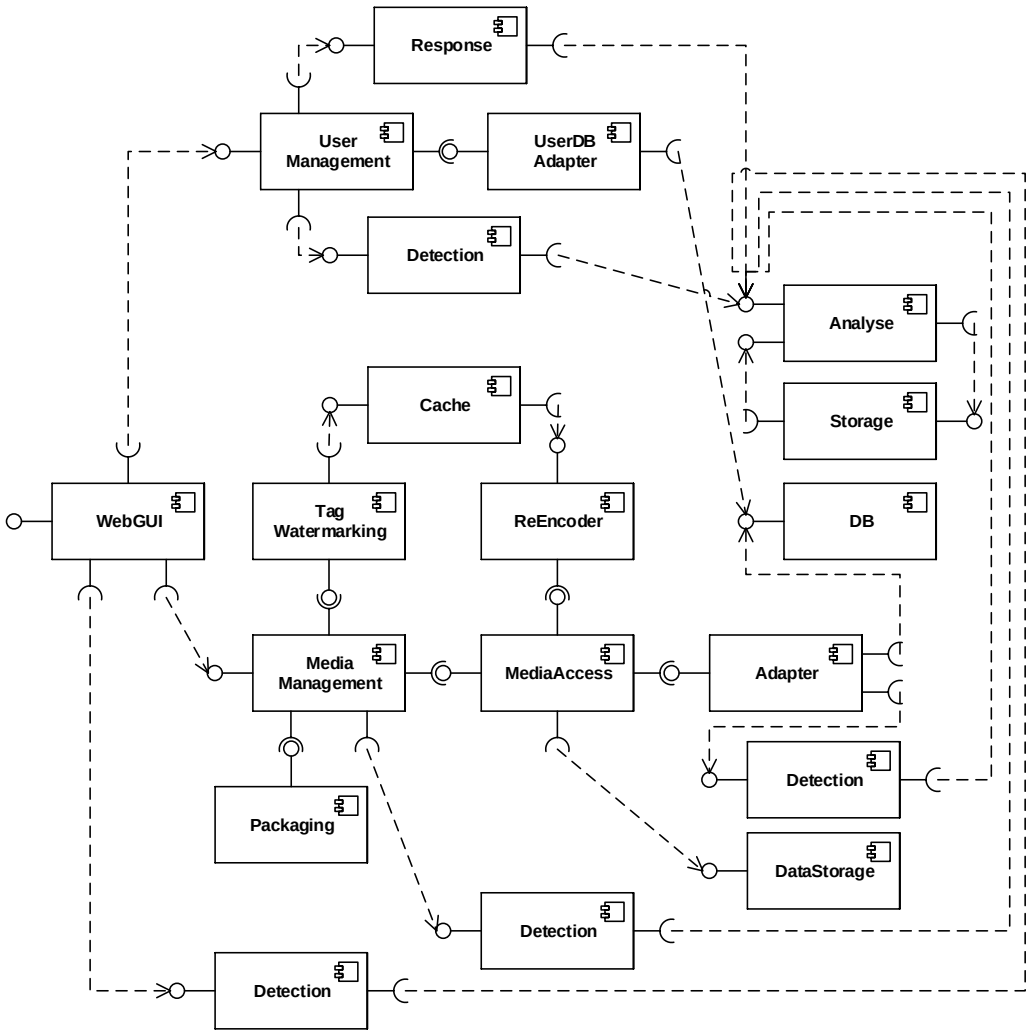


Abbildung 4.13.: Komponentendiagramm des Media Store mit Integration des AppSensor

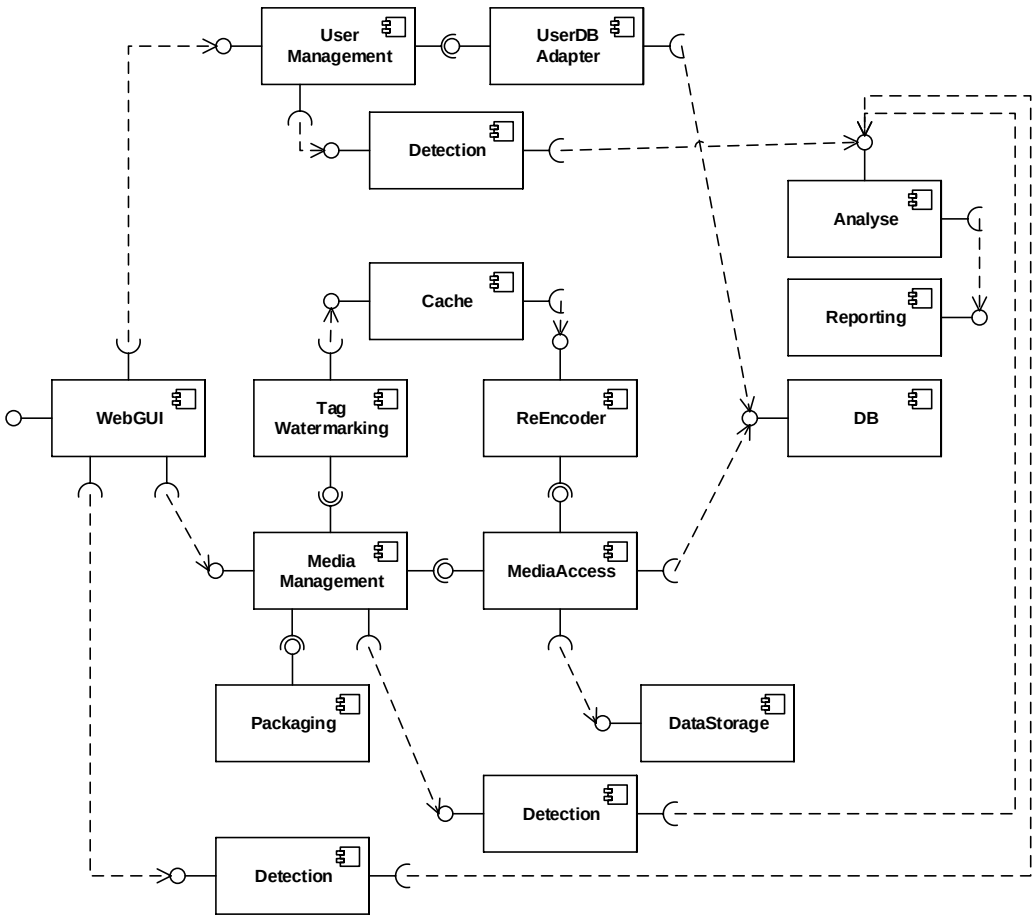


Abbildung 4.14.: Komponentendiagramm des Media Store mit der Integration von OSSEC

5. Zusammenfassung und Ausblick

Im letzten Kapitel wird das Ergebnis dieser Bachelorarbeit zusammengefasst, sowie ein Ausblick auf weitere Erweiterungsmöglichkeiten des Strukturmodells erfolgen.

5.1. Zusammenfassung

In dieser Bachelorarbeit wurde das Strukturmodell als eine Erweiterung für das Palladio Komponentenmodell konzipiert, um mit diesem die Eigenschaften einer Entwurfsentscheidung wiederverwendbar abbilden zu können. Das Strukturmodell umfasst Informationen wie die Struktur einer Entwurfsentscheidung und auch deren Einfluss auf die Qualität eines Softwaresystems. Es wurde gezeigt, wie sich mit Hilfe des Strukturmodells ganze Umbau-Maßnahmen für die Integration der Komponenten einer Entwurfsentscheidung formalisieren lassen.

Hierbei ist der Ansatz, eine Entwurfsentscheidung zusammenhängend sowohl als Provided, Complete und Implementation Type Komponenteninstanz zu modellieren. Der Provided Type definiert die Qualitätseinflüsse einer Entwurfsentscheidung, im Complete Type werden all die Komponenten definiert, welche benötigt werden, um die Entwurfsentscheidungen zu realisieren, und der Implementation Type definiert eine vollständige Implementierung aller von einer Entwurfsentscheidung benötigten Komponenten. Zu den Komponenten einer Entwurfsentscheidung werden zusätzlich Annotationen gespeichert, mit welchen modelliert werden kann, wie das Subsystem einer Entwurfsentscheidung in eine bestehende Architektur zu integrieren ist und auch wie der physische Aspekt der Verteilung dieser Komponenten einer Entwurfsentscheidung beschaffen ist.

Für die Evaluation des Strukturmodells wurden mehrere Fallstudien auf das Strukturmodell angewendet, um so dessen Anwendbarkeit und Nutzen zu zeigen. Hierfür wurde die Struktur von verschiedenen Entwurfsentscheidungen und deren Auswirkungen auf eine bestehende Architektur mit Hilfe des Strukturmodells modelliert und in ein bestehendes Softwaresystem eingesetzt. Es wurde gezeigt, dass durch die Modellierung von Entwurfsentscheidungen mittels des Strukturmodells Entwickler weniger Fachwissen darüber benötigen, wie ein solches Subsystem einer Entwurfsentscheidung in das eigene Softwaresystem zu integrieren ist. Möchte nun ein Entwickler eine bestimmte Entwurfsentscheidung in einem bereits bestehenden Softwaresystem nachträglich umsetzen, ist es für eine modellierte Entwurfsentscheidung einfacher, diese zu integrieren.

5.2. Ausblick

Zum Abschluss dieser Bachelorarbeit soll ein Ausblick auf weitere Themen erfolgen, welche zusätzliche Ansatzpunkte für eventuell nachfolgende Arbeiten behandelt. Es sollen

mehrere mögliche Verbesserungen und Erweiterungen des Strukturmodells kurz erörtert werden, welche im Rahmen dieser Bachelorarbeit nicht mehr umsetzbar waren.

Der Provided-Typ definiert zwar die Qualitätseigenschaften einer Entwurfsentscheidung jedoch der Qualitätseinfluss, welcher durch die Beziehungen der neuen Komponenten mit dem bestehenden System entsteht, wurde nicht modelliert. Eine weitere möglicherweise nachfolgende Arbeit wäre, zunächst architekturelle Freiheitsgrade zu untersuchen und anschließend in Bezug zu ihrem Einfluss auf die Gesamtqualität der Softwarearchitektur zu bringen.

Über die Transformationsobjekte des Strukturmodells werden die grundlegenden Informationen modelliert, welche nötig sind, um zu beschreiben, wo und wie die Komponenten einer Entwurfsentscheidung in ein System zu integrieren sind. Sie beschreiben jedoch noch keine Modelltransformation, um daraus eine neue Instanz des Palladio Komponentenmodells zu generieren. Eine weitere möglicherweise nachfolgende Arbeit wäre, die Erstellung solcher Modelltransformationen aus den bereitgestellten Informationen der Transformationsobjekte.

Mit Hilfe von PCM-Profilen lassen sich Elemente des Palladio Komponentenmodells mit Annotationen erweitern ohne das Metamodell abändern zu müssen. Aktuell werden solche PCM-Profile nur im Standard Baumeditor von Eclipse unterstützt. Eine weitere möglicherweise nachfolgende Arbeit wäre die Integration der Annotationen in den grafischen Sirius-Editor von Eclipse, um damit zum Beispiel Annotationen auf Komponenten ziehen und ablegen zu können.

Das Strukturmodell wurde als Erweiterung des bestehenden Palladio Komponentenmodells konzipiert, um damit in einer späteren Entwicklungsphase (automatische) Qualitätsanalysen durchführen zu können. Eine andere weiterführende Arbeit wäre die vollständige Integration des Strukturmodells in die Palladio-Bench, um somit möglicherweise die resultierende Qualität der Entwurfsentscheidung abhängig von der bereits existierenden Softwarearchitektur analysieren zu können. Auch wäre es denkbar, mit PerOpteryx automatisch die beste Implementierung einer Entwurfsentscheidung für eine bereits existierende Softwarearchitektur zu wählen.

Literatur

[1] ISO/IEC 25010:2011. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO Standard 25010:2011. Genf, Schweiz: International Organization for Standardization, März 2011.

[2] Omar Alam, Jörg Kienzle und Gunter Mussbacher. „Concern Driven Software Development." In: *Demos/Posters/StudentResearch@ MoDELS*. 2013, S. 106–111.

[3] John Cheesman und John Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0-201-70851-5.

[4] Daniel B Cid. „Log analysis using OSSEC". In: *http://www.ossec.net/ossec-docs/auscert-2007-dcid.pdf* (2007).

[5] Software Design, Institute for Program Structures Quality (SDQ) und Karlsruhe Institute of Technology (KIT) Data Organization (IPD). *Media Store SVN-Repository*. `https://svnserver.informatik.kit.edu/i43/svn/code/CaseStudies/MediaStore3`. Apr. 2016.

[6] Holger Eichelberger u. a. „Using IVML to Model the Topology of Big Data Processing Pipelines". SPLC'16 Beijing, P.R. China: Accepted Papers in the research track. 2016.

[7] Palladio Entwicklerteam. *Offizielle Palladio Internetpräsenz*. `http://www.palladio-simulator.com/`. Apr. 2016.

[8] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[9] ITWissen. *Aspektorientierte Programmierung*. `http://www.itwissen.info/definition/lexikon/Aspektorientierte-Programmierung-aspect-oriented-programming-AOP.html`. Apr. 2016.

[10] Ralph E. Johnson und Brian Foote. „Designing reusable classes". In: *Journal of object-oriented programming* 1.2 (1988), S. 22–35.

[11] Gregor Kiczales u. a. „Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Hrsg. von Mehmet Akşit und Satoshi Matsuoka. Bd. 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, S. 220–242. ISBN: 978-3-540-63089-0. DOI: `10.1007/BFb0053381`. URL: `http://dx.doi.org/10.1007/BFb0053381`.

[12] Heiko Koziolek u. a. „Evaluating performance of software architecture models with the Palladio component model". In: *Model-Driven Software Development: Integrating Quality Assurance, IDEA Group Inc* (2008).

[13] Max E Kramer u. a. „Extending the Palladio component model using profiles and stereotypes“. In: *Proceedings of Palladio days* (2012), S. 7–15.

[14] Sebastian Lehrig. „Architectural Templates: Engineering Scalable SaaS Applications Based on Architectural Styles.“ In: *DocSymp@ MoDELS*. Citeseer. 2013, S. 48–55.

[15] Sebastian Lehrig und Max E. Kramer. *SDQ-Wiki: MDSDProfiles*. `https://sdqweb.ipd.kit.edu/wiki/MDSDProfiles`. Nov. 2015.

[16] Anne Martens u. a. „Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms“. In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM. 2010, S. 105–116.

[17] Michael Meier. *Intrusion Detection effektiv!: Modellierung und Analyse von Angriffsmustern*. Bd. 1. Springer-Verlag, 2007.

[18] John Melton. *AppSensorGitHub Repository*. `https://github.com/jtmelton/appsensor`. Sep. 2015.

[19] OWASP. *AppSensor – Application Intrusion Detection and Response*. `http://appsensor.org/`. 2015.

[20] OWASP. *AppSensor Project*. `https://www.owasp.org/index.php/OWASP_AppSensor_Project`. Apr. 2016.

[21] Ralf Reussner. *Component-Based Software Architecture Lecture*. `http://sdqweb.ipd.kit.edu/wiki/KBSWA`. Apr. 2015.

[22] Ralf Reussner und Wilhelm Hasselbring. *Handbuch der Software-Architektur*. 2., überarb. und erw. Aufl. Heidelberg: dpunkt.verlag, 2009. ISBN: 978-3-89864-559-1.

[23] Ralf Reussner u. a. *The Palladio Component Model*. Forschungsbericht ISSN: 2190-4782. Karlsruhe: Chair for Software Design & Quality (SDQ), Karlsruhe Institute of Technology (KIT), Germany, März 2011.

[24] Yves R. Schneider. „Dynamische Instrumentierung: Kieker vs. DiSL“. Proseminararbeit. Karlsruher Institut für Technologie, Juli 2014.

[25] Bundesamt für Sicherheit in der Informationstechnik. *Leitfaden zur Einführung von Intrusion-Detection-Systemen*. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/IDS/Grundlagenv10_pdf.pdf`. Okt. 2002.

[26] Misha Strittmatter und Amine Kechaou. *The Media Store 3 Case Study System*. Karlsruhe Reports in Informatics. Faculty of Informatics, Karlsruhe Institute of Technology, 2016.

[27] OSSEC Project Team. *OSSEC Official Website*. `https://ossec.github.io/`. Apr. 2016.

[28] Colin Watson, Dennis Groves und John Melton. *AppSensor Guide Version 2.0*. Techn. Ber. OWASP Foundation, Juli 2015.

A. Anhang

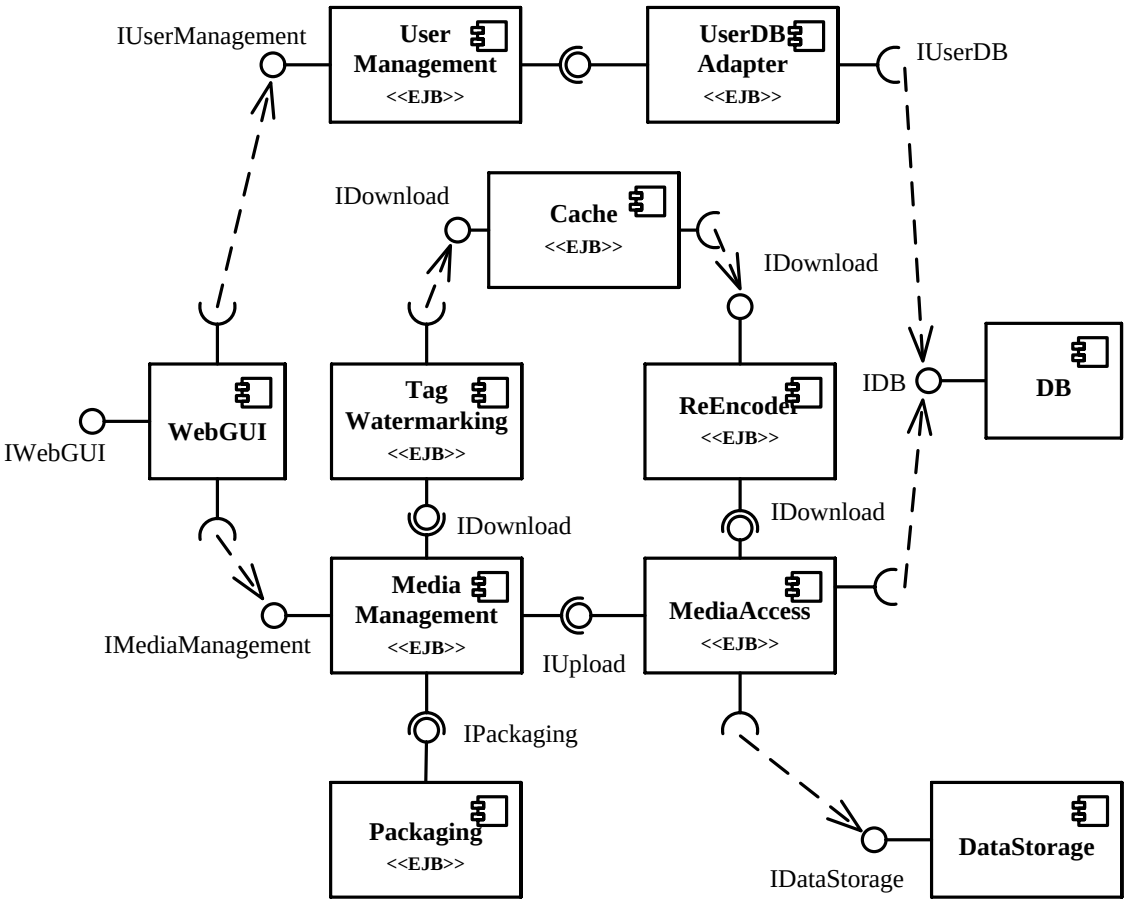


Abbildung A.1.: Original Architekturschaubild des Media Store aus [26, S. 7].

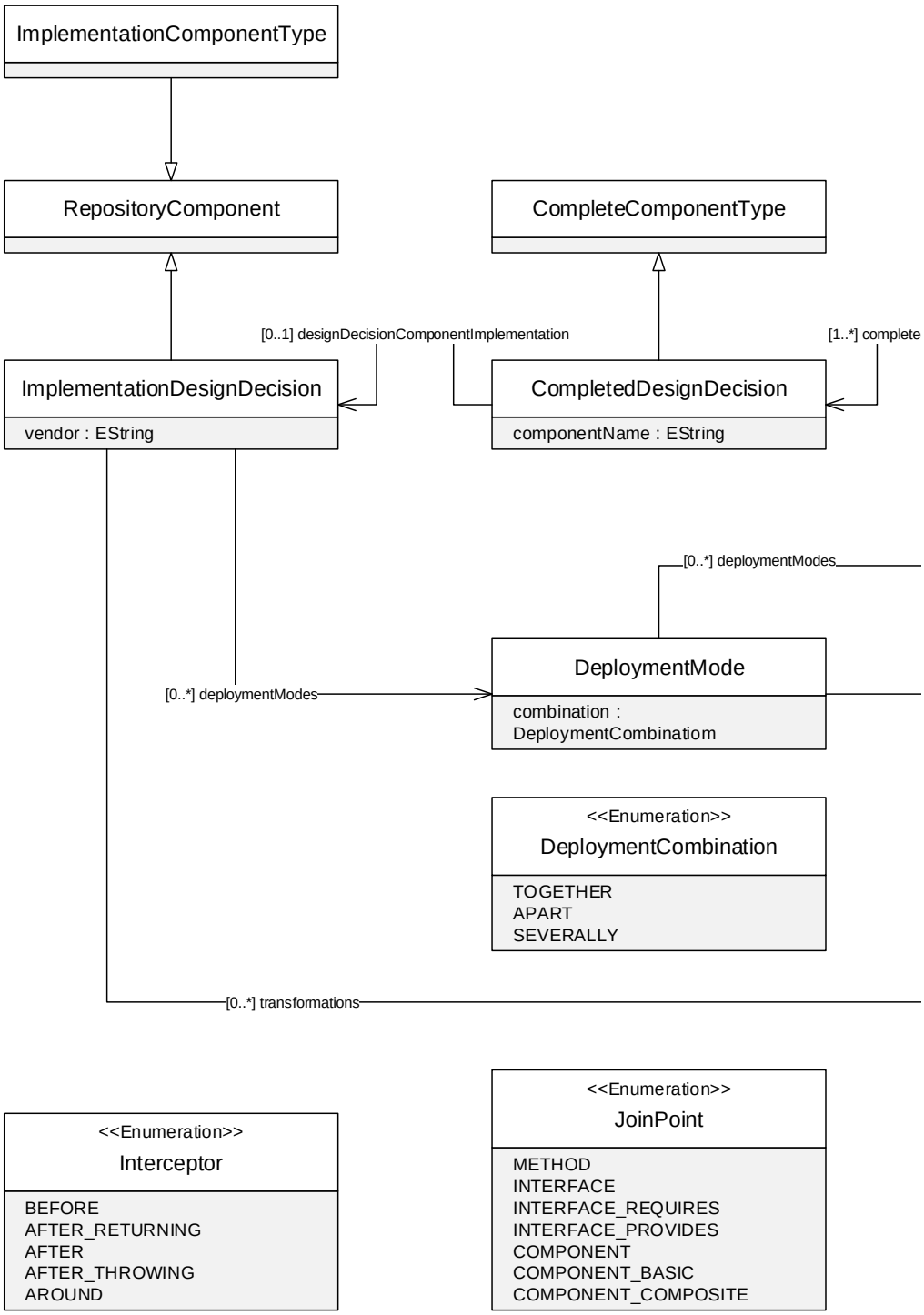
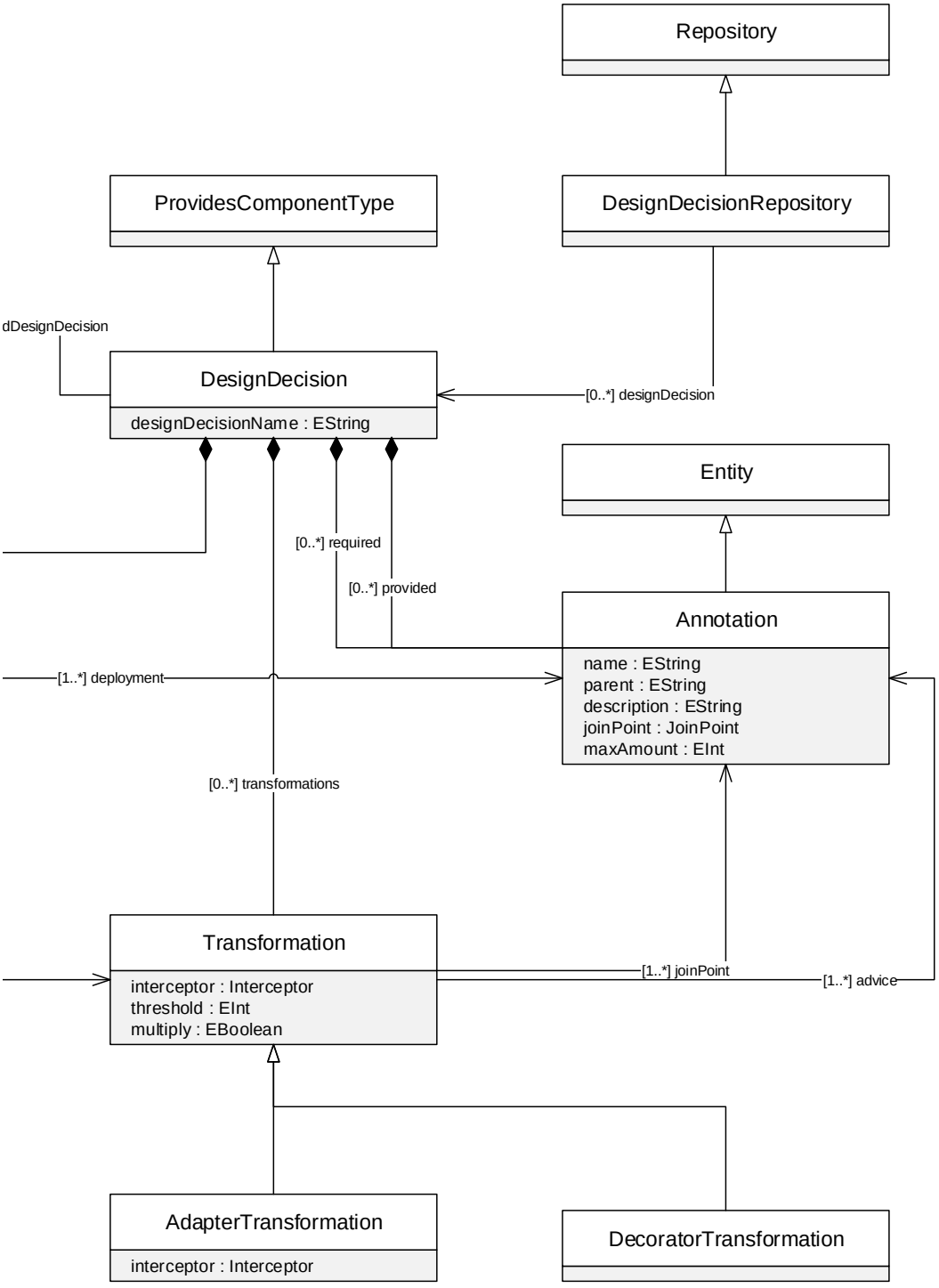


Abbildung A.2.: Strukturmodell


Repository
ProvidesComponentType
DesignDecisionRepository
dDesignDecision
DesignDecision
designDecisionName : EString
[0..*] designDecision
Entity
[0..*] required
[0..*] provided
Annotation
[1..*] deployment
name : EString
parent : EString
description : EString
joinPoint : JoinPoint
maxAmount : EInt
[0..*] transformations
Transformation
interceptor : Interceptor
threshold : EInt
multiply : EBoolean
[1..*] joinPoint
[1..*] advice
AdapterTransformation
interceptor : Interceptor
DecoratorTransformation