Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit

Architekturanalyse und Reengineering einer Prüfumgebung für Spreadsheets

Frieder Schüler

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Dr. Ivan Bogicevic, Daniel Kulesz, M.Sc.
Beginn am:	29. September 2016
Beendet am:	31. März 2017
CR-Nummer:	D.2.7, D.2.11

Kurzfassung

In dieser Arbeit werden verschiedene Methoden, Werkzeuge und Prozesse untersucht, die dazu genutzt werden können, die Qualität der Software im Rahmen eines Reengineerings zu verbessern. Am Beispiel des Spreadsheet Inspection Framework (SIF), das an der Universität Stuttgart entwickelt und zur Prüfung von Spreadsheets („Excel-Programme“) genutzt wird, werden diese Verfahren angewendet und ein Reengineering durchgeführt. Bei der Auswahl der Verfahren wird der Fokus auf die Analyse und Bewertung der Softwarearchitektur gelegt.

Mit Hilfe der szenariobasierten Analyseverfahren Software Architecture Analysis Method (SAAM) und Architecture Level Modifiability Analysis (ALMA), der Betrachtung verschiedener Metriken und der Analyse der Dependency Structure Matrix (DSM) werden die Probleme der Architektur identifiziert und anschließend überarbeitet. Die Ergebnisse einer Architekturanalyse der überarbeiteten Software zeigen dabei, dass die Softwarequalität in vielen Punkten gesteigert, die Komplexität der Architektur reduziert und der Umfang des Quellcodes halbiert wird. Abschließend wird der Erfolg der verwendeten Methoden und Werkzeuge diskutiert, bewertet wie die einzelnen Ergebnisse der Verfahren genutzt werden und festgestellt ob der Einsatz dieser Verfahren für ein Reengineering geeignet ist.

Abstract

This paper addresses the analysis of methods, tools and processes which can be used in software maintenance to increase the software quality. The software which will be reengineered is the Spreadsheet Inspection Framework (SIF). This software is developed at the University of Stuttgart and is used to check spreadsheets. The focus of this work will be the evaluation of the software architecture.

The scenario-based analysis methods SAAM and ALMA, the evaluation of different metrics and the analysis of the Design Structure Matrix will be used to identify the weak spots of the software architecture. These components will be redesigned and replaced by a new architecture. The architecture analysis of the improved software shows that the software quality has improved, the architecture's complexity was reduced and the amount of source code was shortened by 50%. Afterwards the analysis techniques and tools applied here will be evaluated and a rating for usefulness of the methods will be given.

Inhaltsverzeichnis

1 Einleitung

Es ist seit langem bekannt, dass auch Software altert und regelmäßige Anpassungen im Rahmen einer Softwarewartung benötigt. Je nach Qualität und Umfang der Software kann diese Wartung einen erheblichen Teil der Softwarekosten verursachen. In dieser Arbeit wird untersucht wie die Qualität einer Software, innerhalb einer Softwarewartung, mit möglichst geringen Kosten verbessert werden kann. Schwerpunkt der Untersuchungen ist dabei die Softwarearchitektur, da diese großen Einfluss auf die einzelnen Qualitätsmerkmale der Software hat.

1.1 Motivation

Bei der im Rahmen dieser Arbeit untersuchten Software handelt es sich um das Spreadsheet Inspection Framework (SIF), eine Software zur Prüfung von Spreadsheets. Das Grundgerüst sowie alle Erweiterungen der Funktionalität von SIF wurden im Rahmen von studentischen Abschlussarbeiten entwickelt. Diese Arbeit an SIF fand dabei, mit Unterbrechungen, über einen Zeitraum von sieben Jahren statt. Jede der Entwicklungs- oder Wartungsphasen wurde von einem anderen Entwickler durchgeführt und es gab dabei keinen einheitlichen Entwicklungsprozess. Außerdem konnten die einzelnen Entwickler nur selten miteinander kommunizieren und mussten sich jeweils selbstständig in die Arbeit ihrer Vorgänger einarbeiten.

Unter diesen Bedingungen ist es verständlich, dass sich die Software im Moment in einem nicht zufriedenstellenden Zustand befindet. Der Projektleiter von SIF beklagt dabei verschiedene Probleme, insbesondere hinsichtlich der Qualitätsmerkmale Zuverlässigkeit und Wartbarkeit. Er vermutet aufgrund der unsteten Entwicklungsgeschichte von SIF auch eine hohe Erodierung der Architektur.

1.2 Ziele

Das Ziel dieser Arbeit ist es, Methoden und Werkzeuge zu finden und zu untersuchen, die in einem Wartungsprozess verwendet werden können, um die Softwarequalität des Spreadsheet Inspection Frameworks zu verbessern. So sollen die bisherigen Probleme von SIF behoben und gleichzeitig eine zukünftige Wartung der Software vereinfacht werden. Es sollen vorrangig Methoden oder Werkzeuge zur Evaluation von Softwarearchitekturen verwendet werden, weil schwerpunktmäßig die Architektur der Software analysiert, bewertet und verbessert werden soll. Mit den aus dieser Evaluation gewonnenen Erkenntnissen sollen die Schwachstellen in der Softwarearchitektur mit einem neuen Architekturentwurf behoben werden. Dabei soll die Qualität der Software und

besonders die Qualitätsmerkmale Wartbarkeit und Zuverlässigkeit verbessert werden. Der Erfolg dieser Maßnahmen soll dann durch entsprechende Tests oder Evaluationsschritte überprüft und dokumentiert werden. Zum Abschluss sollen die zur Anwendung gekommenen Methoden oder Verfahren dann selbst auf ihre Aussagekraft und Nützlichkeit hin untersucht und bewertet werden.

1.2.1 Untersuchte Fragen

Um die Untersuchung der Verfahren zur Architekturanalyse, sowie die damit verbundenen Ziele, noch konkreter zu benennen, werden die folgenden Fragen formuliert, die dann im Laufe der Arbeit beantwortet werden sollen:

1. Welche Verfahren sind für ein Reengineering hilfreich?
2. Welche Aspekte untersuchen die Verfahren?
3. Wie unterscheiden sich die Ergebnisse dieser Verfahren?

1.2.2 Abgrenzung

Während die bisherigen Entwicklungs- und Wartungsphasen hauptsächlich dafür gesorgt haben, dass der Software neue Prüffunktionen für Spreadsheets hinzugefügt wurden, sollen diese Änderungen nun zusammengeführt, überarbeitet und verfeinert werden, um die Softwarequalität zu verbessern. Auch liegen die Schwerpunkte der bisherigen Arbeiten darauf, wie Spreadsheets geprüft oder wie solche Prüfergebnisse visualisiert werden können. In dieser Arbeit werden deshalb weder neue Prüfmöglichkeiten für Spreadsheets entwickelt noch die Prüfung von Spreadsheets näher untersucht. Anpassungen am Visualisierungswerkzeug SIFEI sollen auch nur vorgenommen werden, wenn Änderungen an SIFCore diese erforderlich machen. Abgesehen davon sind Verbesserungen von SIFEI nicht Teil dieser Arbeit.

1.2.3 Beitrag

Für die Wiederherstellung der Architektur wird ein geeignetes Verfahren konzipiert und anschließend umgesetzt. Es werden Verfahren aus verschiedenen Kategorien der Architekturbewertung untersucht, um mit diesen die bisherige Softwarearchitektur zu analysieren und zu bewerten. Dabei werden die kritischen Schwachstellen in der Architektur identifiziert und im Rahmen eines Reengineerings behoben. Es werden die Basisfunktionalität und die bisher erfolgten Erweiterungen von SIFCore in eine neue Architektur integriert, um die Qualität der Software zu verbessern. Im Rahmen einer abschließenden Evaluation werden die eingesetzten Bewertungs- und Analyseverfahren selbst kritisch bewertet.

1.3 Gliederung

Diese Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Beschreibt die Grundlagen, der in dieser Arbeit verwendeten Methoden und Prozesse.

Kapitel 3 – Architekturrekonstruktion: Beschreibt das Verfahren sowie die Ergebnisse der Architekturrekonstruktion.

Kapitel 4 – Architekturanalyse: Behandelt die Auswahl, das Tailoring und die Durchführung der genutzten Analyseverfahren.

Kapitel 5 – Anforderungen: Legt die Anforderungen an die verbesserte Architektur fest.

Kapitel 6 – Architekturentwurf: Beschreibt die verbesserte Architektur und die Hintergründe, die zu diesen Entwurfsentscheidungen geführt haben.

Kapitel 7 – Implementierung: Beschreibt die konkrete Umsetzung der Reengineering-Maßnahmen.

Kapitel 8 – Architekturanalyse II: Behandelt die erneute Analyse der verbesserten Architektur und diskutiert anschließend die Unterschiede zwischen beiden Architekturen.

Kapitel 9 – Evaluation der Analyseverfahren: Analysiert die Kosten und den Nutzen der eingesetzten Bewertungsverfahren.

Kapitel 10 – Fazit: Fasst die Ergebnisse der Arbeit zusammen und gibt einen kurzen Ausblick auf mögliche zukünftige Arbeiten.

2 Grundlagen

2.1 Softwarewartung

Der Begriff der Softwarewartung umfasst mehr als man auf den ersten Blick denkt. Das liegt daran, dass im englischen der Begriff „software maintenance“ verwendet wird und dieser im Deutschen mit „Wartung“ übersetzt wird. Der Begriff der Wartung hat im deutschen Sprachgebrauch allerdings eher die Bedeutung der Reparatur, wird also hauptsächlich verwendet, wenn etwas bereits defekt und nicht mehr funktionsfähig ist und deshalb gewartet werden muss. Im Englischen wird „maintenance“ allerdings im Sinne von Instandhaltung, Versorgung oder Betreuung verwendet und impliziert also keinesfalls, dass etwas defekt ist.

Softwarewartung wird der IEEE als „die Veränderung einer Software nach Auslieferung, um Fehler zu beheben, die Performanz oder andere Attribute zu verbessern oder an eine veränderte Umgebung anzupassen“ [Ieeb]. Swanson et al. [LS80] haben gezeigt, dass die Wartung von Software nicht nur der arbeitsintensivste Abschnitt im Software-Lebenszyklus ist, sondern dass es unterschiedliche Motivationen für eine Softwarewartung gibt. Nach Swanson et al. [LS80] können Veränderungen während der Wartung in die folgenden Kategorien eingeordnet werden:

- Es werden Korrekturen vorgenommen (*corrective maintenance*) um bereits bestehende Fehler zu beseitigen.
- Es werden Anpassungen durchgeführt (*adaptive maintenance*) um auf Änderungen der Umgebung, zum Beispiel des Betriebssystem, reagieren zu können.
- Es werden Funktionen hinzugefügt (*perfective maintenance*) um auf geänderte Anforderungen zu reagieren.
- Es werden Veränderungen vorgenommen (*preventive maintenance*) um die Wartbarkeit zu erhöhen und latente Fehler zu vermeiden, bevor diese als echte Fehler auftreten.

Gründe für die Softwarewartung sind also geänderte Anforderungen an die Software, Änderungen an der Umgebung des Softwaresystems, die Behebung von neu entdeckten Fehlern oder das Vereinfachen der zukünftigen Wartungsarbeiten. Seit sich Ende der 1970er Jahre die Informatik erstmals genauer mit dem Wartungsprozess von Software auseinandersetzte, wurde klar, dass die Wartung von Software einen erheblichen Teil der Kosten verursacht und einen großen Anteil der Entwicklungszeit beansprucht. Auswertungen wie die von Swanson et al. [LS80] sprechen von 50-80% der gesamten Entwicklungszeit. Dabei entfallen 51% der aufgewendeten Zeit auf „perfective“ Änderungen, 24% auf „adaptive“ Änderungen, 22% auf „corrective“ Änderungen und nur 3% sind „preventive“ Änderungen. Dass nur 3% der Arbeiten der letzten Kategorie zugeordnet wurden, liegt eventuell auch daran, dass diese Definition recht schwammig und laut Ludewig

et al. [Lud13] sogar irreführend ist. Die Autoren kritisieren, dass die präventive Wartung in Wirklichkeit auch nur eine Korrektur ist, da es keine „latenten Fehler“ gibt. Entweder liegt ein Fehler vor, das bedeutet die Software entspricht nicht der Spezifikation, oder es liegt kein Fehler vor, wenn die Software sich gemäß der Spezifikation verhält. Die Untersuchung von Swanson wurde 1990 von Nosek et al. [NP90] wiederholt und diese kamen dabei auf eine sehr ähnliche Verteilung, bestätigen also die Studie von Swanson.

Da Software im Gegensatz zu materiell existierenden Dingen nicht im herkömmlichen Sinne altert verändert sich der Zustand der Software, ohne absichtliche Änderungen der Software oder der davon abhängigen Systemen, natürlich nicht. Alle Fehler sind also schon im Originalzustand vorhanden. Da sich die Umgebung, in der die Software betrieben wird, ändern kann oder sich die Anforderungen an die Software wandeln, wird auch ohne Veränderung der Software eine Verschlechterung der Nützlichkeit eintreten. Diese wurde schon früh von Lehmann et al. [Leh80] beobachtet und im „Law of Continuing Change“ beschrieben. So muss ein „e-type programm“, eine Software die in andere Systeme eingebettet ist oder mit anderer Software in Wechselwirkung steht, ständig angepasst werden. Ansonsten werden sich Nutzen und Funktionalität der Software immer weiter verschlechtern. Später aktualisierten Lehmann et al. [Leh96] ihre bisherigen Untersuchungen und formulierten das „Law of Declining Quality“, das besagt, dass die Qualität der Software mit steigender Lebensdauer immer weiter abnimmt, vorausgesetzt natürlich, dass nichts aktiv dagegen unternommen wird. Diese Beobachtungen wurden zum Beispiel durch Yu et al. [YM13] bestätigt. In ihrer Untersuchung der Open-Source Software Apache Tomcat[1] und Apache Ant[2] bestätigt sich diese These, nach Meinung der Autoren, zumindest in diesen Fällen.

Die Softwarewartung soll der abnehmenden Nützlichkeit und Qualität der Software aktiv entgegenwirken und diese verbessern oder zumindest erhalten. Wie erfolgreich eine Softwarewartung ist, also ob und wie die Softwarequalität gesteigert werden kann, hängt von verschiedenen Faktoren ab. Lientz et al. [LS81] identifizierten sechs Faktoren, die den größten Einfluss während der Wartung haben. Dabei haben sie als wichtigsten Faktor den Wissensstand der Entwickler identifiziert. Laut ihrer Untersuchung liegt der Anteil dieses Faktors bei 60%. Ob eine Wartung Erfolg hat, hängt also stark von den Entwicklern ab und weniger von anderen Faktoren wie die verfügbare Zeit oder die Qualität der Software.

2.2 Reengineering

Der Begriff des *Reengineering*, was auf deutsch am ehesten mit „ingenieurmäßigem Überarbeiten“ übersetzt werden kann, wird von Chikofsky et al. [CI90] wie folgt definiert: Die Prüfung und Veränderung eines Systems mit dem Ziel es mit einer neuen, verbesserten Form zu ersetzen. Das *Reengineering* ist dabei die Kombination von *Reverse Engineering* und *Forward Engineering*. Ersteres ist die Analyse und Dokumentation eines bis dahin unbekannten oder undokumentierten Systems. Letzteres ist der traditionelle Prozess des Architekturentwurfs und die anschließende Umsetzung (Implementierung) dieses Entwurfs. Das *Reengineering* ist also eine Spezialisierung

[1] http://tomcat.apache.org/
[2] http://ant.apache.org/

der Softwarewartung, bei der nicht nur kleinere Änderungen, sondern eine vollständige Überarbeitung der Software das Ziel ist.

2.3 Softwarequalität

Der Begriff der Softwarequalität ist zusammengesetzt aus den Begriffen Software und Qualität. Während die Definition von Software recht einfach gelingt und weitestgehend dem allgemeinen Sprachgebrauch entspricht, ist das bei Qualität so nicht der Fall. Im alltäglichen Sprachgebrauch wird Qualität oft im Sinne von „guter Qualität“ verwendet, dabei ist bei der Verwendung des Begriffes keine Wertung impliziert [Dud]. In der Norm [Iso] wird **Qualität** als „Grad, in dem ein Satz inhärenter Merkmale eines Objekts Anforderungen erfüllt“ definiert. *Inhärent* bedeutet dabei dem Objekt innewohnend, das sind alle objektiv messbaren Eigenschaften eines Objekts. Die Qualität wird also dadurch bestimmt, in wie weit die einzelnen **Qualitätsmerkmale** erfüllt sind. Diese Definition von Qualität kann nun direkt auf die Softwarequalität übertragen werden, allerdings ist dabei oft unklar welche Qualitätsmerkmale für Software definiert werden sollten.

Um herauszufinden welche Qualitätsmerkmale für Softwarequalität sinnvoll sind, muss zunächst noch einmal der Qualitätsbegriff genauer untersucht werden. So ist für die Entwickler bei der Softwareentwicklung nicht nur die Qualität des fertigen Produkts (also der Software) entscheidend, sondern auch die Qualität des Herstellungsprozesses. Wenn letzteres nicht kontrolliert und gesteuert wird, entsteht mit Glück vielleicht eine brauchbare Software, allerdings mit hohen Kosten. Es muss also zwischen Prozessqualität und Produktqualität unterschieden werden.

Die Prozessqualität beschreibt die Qualität des Prozesses mit dem das Produkt hergestellt wird. Sie bestimmt, wie weit es der Hersteller schafft Kosten- und Terminvorgaben einzuhalten, Kenntnisse zu sammeln oder zum Beispiel wiederverwertbare Komponenten zu erstellen, um diese in einem späteren Produkt wiederverwenden zu können. Prozess- und Produktqualität hängen zusammen. Eine hohe Prozessqualität fördert dabei eine hohe Produktqualität, kann diese aber nicht erzwingen. Umgekehrt hat eine schlechte Prozessqualität nicht direkt eine schlechte Produktqualität zur Folge, macht diese aber wahrscheinlicher.

Die Produktqualität beschreibt die Qualität des eigentlichen Produktes, in diesem Fall also der Software. Die Produktqualität kann noch weiter in Gebrauchs- und Wartungsqualität unterteilt werden. Die Gebrauchsqualität beinhaltet dabei Qualitätsmerkmale, die wichtig für den Benutzer des Produkts sind, in diesem Fall also der Benutzer der mit der Software arbeitet. Dazu gehören die Qualitätsmerkmale wie Performanz, Zuverlässigkeit und Bedienungsfreundlichkeit. Die Wartungsqualität ist dagegen für denjenigen wichtig der an der Software arbeitet, sie also korrigieren oder erweitern muss. Hier sind Qualitätsmerkmale wie Lesbarkeit, Veränderbarkeit oder Portabilität wichtig.

Es fällt also auf, dass der Qualitätsbegriff nicht so eindeutig ist wie es zuerst erscheinen mag, so ist die Definition des Qualitätsbegriffs immer abhängig von der Perspektive des Betrachters. [Gar84] nennt fünf mögliche Sichtweisen für den Qualitätsbegriff:

Transzendente Sicht
Bei der transzendenten Sicht wird Qualität als unmöglich zu erreichendes Ideal verstanden. Man kann sich diesem Ideal annähern und sollte auch versuchen es zu erreichen, aber die Qualität kann nie exakt definiert und gemessen werden.

Benutzerorientierte Sicht
Bei der benutzerorientierten Sicht entscheiden die Nutzer und Konsumenten subjektiv über die Qualität eines Produkts. Damit hängt die Definition von Qualität immer vom jeweiligen Nutzer oder Nutzergruppe und dem Kontext ab. Für die Nutzer sind vor allem die Qualitätsmerkmale Funktionalität, Zuverlässigkeit und Benutzbarkeit wichtig. Auch betrachtet der Nutzer fast ausschließlich die Produktqualität und kümmert sich nur in den wenigsten Fällen um die Prozessqualität.

Herstellerorientierte Sicht
Bei der herstellerorientierten Sicht wird die Qualität aus Sicht des Herstellers bewertet. Eine hohe Qualität bedeutet bei dieser Sicht die Einhaltung der Vorgaben bei der Entwicklung, hauptsächlich in Bezug auf Kosten und Terminvorgaben.

Produktorientierte Sicht
Bei der produktorientierten Sicht wird Qualität als exakt definierbare und auch auch messbare Größe verstanden. So lässt sich die Qualität eines Produkts bestimmen und kann mit anderen Produkten verglichen werden. Im Gegensatz zur benutzerorientierten Sicht wird die Qualität objektiv und kontextfrei bewertet.

Wertorientierte Sicht
Bei der wertorientierten Sicht wird Qualität über die Kosten des Produkts verbunden mit verschiedenen Eigenschaften definiert. Eine hohe Qualität bedeutet in diesem Fall ein gutes Kosten-Nutzen-Verhältnis. So werden zum Beispiel Produkttests in Zeitschriften meist aus der wertorientierten Sicht bewertet.

2.3.1 Taxonomie der Softwarequalität

Nachdem nun die Softwarequalität aus verschiedenen Sichten betrachtet und schon festgestellt wurde, dass es eine Vielzahl von Qualitätsmerkmalen gibt, die zur Softwarequalität gehören, müssen diese in eine Taxonomie überführt werden. Dazu werden nun möglichst disjunkte Definitionen der einzelnen Qualitätsmerkmale gesucht. In dieser Arbeit wird dabei der von Ludewig et al. [Lud13] definierte „Qualitätenbaum“ verwendet. So soll sichergestellt sein, dass keine Qualitätsmerkmale bei der Bewertung der Softwarequalität vergessen werden und die Definitionen der einzelnen Qualitätsmerkmale auch eindeutig sind.

2.4 Softwarearchitektur

Der Begriff der Softwarearchitektur (oder auch nur Architektur) wird im Bereich des Software Engineerings sehr häufig verwendet. Dennoch gibt es keinen Konsens über die exakte Definition

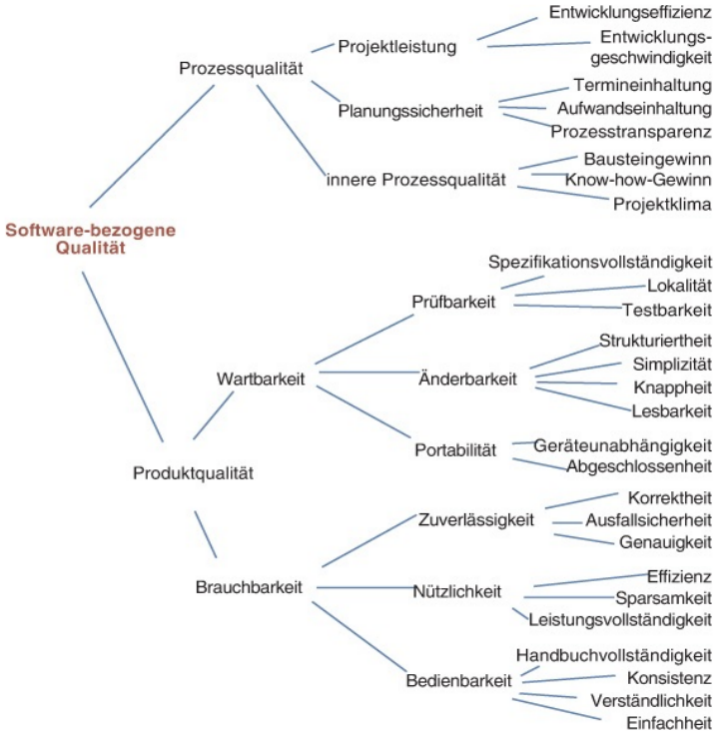

Abbildung 2.1: Qualitätenbaum aus [Lud13]

dieses Begriffes und es existieren viele verschiedene Definitionen [BCK03]. In dieser Arbeit wird der Begriff im Sinne des IEEE Standards 1471-2000 [Ieea] verwendet, dessen Definition „*architecture: fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*“ wie folgt übersetzt und interpretiert wird: Die Architektur ist die Gesamtstruktur eines (Software-) Systems, gegliedert in einzelne Komponenten sowie deren wesentlichen Merkmale und Schnittstellen. Die Schnittstellen definieren dabei Beziehungen zu anderen Komponenten des Systems oder zur Umgebung der Software. Zusätzlich gehören zur Architektur auch die Prinzipien und Intentionen, die bei der Entwicklung der Software angewendet oder verfolgt wurden.

Auf die Struktur der Architektur gibt es verschiedene Sichten, die sowohl dynamische als auch statische Aspekte umfassen können. Die Architektur ist also nicht nur als (statischer) Bauplan für die Software zu verstehen, der die Komponenten und ihre Beziehungen beschreibt, sondern umfasst zum Beispiel auch, wie diese Komponenten beim Ausführen (dynamisch) zusammenarbeiten. Die Architektur ist also auch Grundlage für die Softwarequalität, denn sie legt fest, zu welchem Grad die nicht-funktionalen Anforderungen (und damit die Qualitätsmerkmale) erfüllt werden. Die Architektur einer Software verändert sich durch Entscheidungen des Architekten oder Entwicklers. Sie ist also das Resultat aller Entscheidungen, die während des Softwareentwurfs getroffen wurden. Aus diesem Grund ist es auch wichtig, alle Entscheidungen während des Architekturentwurfs systematisch durchzuführen und entsprechend zu dokumentieren.

2.4.1 Architekturbeschreibung

Der Definition nach ist die Architektur eine abstrakte Sammlung unterschiedlicher Sichten. Ein Entwickler kann jedoch nur mit konkreten Dingen arbeiten, deshalb wird noch der Begriff der Architekturbeschreibung eingeführt. Die Architekturbeschreibung ist die Sammlung aller real existierender Artefakte, die die Architektur beschreiben beziehungsweise dokumentieren. Dieser Unterschied zwischen der (abstrakten) Architektur und der (realen) Architekturbeschreibung ist in der Praxis durchaus relevant. So besitzt jede Software eine Architektur, egal ob sie beschrieben wird oder nicht. Die Beschreibung dieser Architektur kann aber unvollständig, falsch oder nicht vorhanden sein. Dies hat zur Folge, dass Entwickler die Architektur einer Software nicht so einfach erfassen können, oder diese falsch interpretieren. Warum während des Softwareentwurfs eine bestimmte Entscheidung getroffen wird und die Architektur deshalb eine bestimmte Eigenschaft besitzt, ist oft ein Detail, das für spätere Entwickler wichtig ist. Leider zeigt die Erfahrung, dass gerade diese Entscheidungen nicht dokumentiert werden und somit nur schwer nachzuvollziehen sind. Ohne die Architektur einer Software verstanden zu haben, können Änderungen an der Software nur mit dem *Trial and error* Verfahren getestet werden, es kann also nicht abgeschätzt werden, welche Folgen die Änderung für die Softwarearchitektur haben wird. Es ist also im Interesse aller Entwickler, dass die Architekturbeschreibung möglichst vollständig, verständlich und auch korrekt ist.

2.4.2 Architektur- und Entwurfsmuster

Um die Erstellung von Software zu beschleunigen, ist es gängige Praxis bereits implementierte Komponenten aus anderen Projekten erneut zu verwenden. Damit Teile einer Architektur wiederverwendet werden können, werden sogenannte Architektur- und Entwurfsmuster verwendet. Dabei beschreiben beide Konzepte die gleiche Idee, nämlich Lösungsschablonen für den Architekturentwurf zu definieren, die dann ohne großen Aufwand für wiederkehrende Probleme verwendet werden können. Das Architekturmuster, teilweise auch Architekturstil genannt, unterscheidet sich vom Entwurfsmuster nur in der Abstraktionsebene auf der es angewendet wird. Architekturmuster lösen in der Regel Probleme der oberen Abstraktionsebenen, während sich Entwurfsmuster oft auf konkrete Objekte von Komponenten beziehen und sich deshalb deutlich näher an der Implementierung positionieren.

Diese Idee der Architekturmuster wurde zuerst von Alexander et al. [Ale+77] beschrieben und später von Cunningham et al. [CB89] aufgegriffen und verfeinert. Erst ab den 90er Jahren und mit der zunehmenden Popularität von objektorientierten Programmiersprachen, wurden Architektur- und Entwurfsmuster von der Forschung intensiver betrachtet. Diese Arbeit stützt sich vor allem auf die Erkenntnisse von Gamma et al., die in ihrem Buch „*Design Patterns: Elements of Reusable Object-Oriented Software*“ [Gam+95] Entwurfsmuster intensiv behandeln. Dort werden die Entwurfsmuster im Rahmen eines Musterkatalogs klassifiziert und beschrieben, damit mit geringem Aufwand das geeignete Entwurfsmuster für ein bestimmtes Problem gefunden werden kann.

Serviceorientierte Architektur

Die Service-oriented Architecture (SOA) ist ein Architekturmuster, das die Verteilung einer Software in verschiedene Komponenten propagiert, die über ein Netzwerk miteinander kommunizieren. Die OpenGroup definiert in ihrem SOA Source Book [Ope17] wie folgt: „*Service-Oriented Architecture is an architectural style that supports service-orientation.*“ Dabei ist mit *serviceorientiert* die Verwendung von *Services* gemeint. Ein Service zeichnet sich durch die folgenden Eigenschaften aus:

- Er repräsentiert einen (wiederholbaren) Geschäftsprozess mit einem definierten Ergebnis.
- Er ist selbständig und in sich abgeschlossen.
- Er ist für den Benutzer eine Black-Box.
- Er kann eine Komposition mehrerer anderer Services sein.

Neben dem Buch der OpenGroup gibt es noch ein weiteres Buch von Erl et al. [Erl+12], das die Umsetzung einer SOA mit *RESTful Services* behandelt und auf dem teilweise der Architekturentwurf von SIFCore nach der Überarbeitung basiert. Representational State Transfer (REST) beschreibt dabei ein Programmierparadigma für die Umsetzung von Webservices. Es wurde 2002 von Fielding et al. [FT02] veröffentlicht und fordert die Beachtung von sechs Prinzipien für die Umsetzung von Systemen, die *restful* sind. Dazu gehören die Zustandslosigkeit von Nachrichten, das einheitliche Design der angebotenen Schnittstellen und die Umsetzung als Client-Server Architektur. Es ist ein erklärtes Ziel von Fielding et al., dass sich REST möglichst einfach in die schon bestehende Web-Infrastruktur eingliedern lässt.

REST hat im Gegensatz zu Simple Object Access Protocol (SOAP), das oft für die Umsetzung einer SOA verwendet wird, keine umfangreiche und komplizierte Spezifikation, sondern setzt darauf, dass der Entwickler unter Beachtung der REST-Prinzipien etablierte und weit verbreitete (Web-)Technologien einsetzt. REST schreibt zum Beispiel kein Protokoll zur Kommunikation zwingend vor, es konnten im Rahmen der Recherche jedoch nur Softwareprojekte gefunden werden die das Hypertext Transfer Protocol (HTTP) beziehungsweise das Hypertext Transfer Protocol Secure (HTTPS) Protokoll verwenden. Auch bieten die meisten modernen Programmiersprachen, wie in diesem Fall Java, bereits Systembibliotheken an, mit denen sich ein RESTful Webservice einfach umsetzen lässt.

Dependency Injection

Dependency Injection (DI) ist ein Entwurfsmuster in der objektorientierten Programmierung, das von Fowler et al. [Fow04] eingeführt wurde. Er setzt damit das *Dependency Inversion Principle (DIP)*, also das Abhängigkeits-Umkehr-Prinzip, um. Diese von Martin et al. [Mar03] beschriebene Prinzip sagt aus, dass Module höherer Ebenen nicht von Modulen niedrigerer Ebenen abhängen sollten und dass diese Module nur von Abstraktionen abhängen sollten. Gleichzeitig sollten Abstraktionen nicht von Details abhängen, sondern Details sollten von Abstraktionen abhängen.

Normalerweise werden die von einem Objekt benötigten Abhängigkeiten während der Initialisierung des Objekts selbst erzeugt. Dazu benötigt das Objekt aber Informationen über seine Umgebung, um die entsprechenden Abhängigkeiten (und wiederum ihre Abhängigkeiten) zu erzeugen. Das führt jedoch meist zu einer Verletzung des DIP, die mit Hilfe der DI vermieden werden kann: Es wird der Aufbau des Abhängigkeitsgraphen aus den einzelnen Komponenten ausgelagert und in eine zentrale, einzelne Komponente überführt. Diese ist global für die Erzeugung der benötigten Objekte verantwortlich und kann diese bei Bedarf *injizieren*. Dies verringert die Kopplung zwischen Modulen und sichert zu, dass das Single Responsibility Principle (SRP) eingehalten wird, indem die Erstellung der Objekte und die Geschäftslogik getrennt werden.

2.4.3 Architekturrekonstruktion

Der Begriff der Architekturrekonstruktion wird in dieser Arbeit als Übersetzung des englischen *„Software architecture recovery“* verwendet. Er beschreibt dabei laut [Ieea] einen Prozess, der das Ziel hat, die Architekturbeschreibung einer Software in einen zufriedenstellenden Zustand zu bringen. Es ist also in diesem Sinne mehr eine Rekonstruktion der Architekturbeschreibung, da man argumentieren kann, dass die Architektur einer Software immer existiert, unabhängig davon ob diese Architektur nun in einer Form dokumentiert und beschrieben ist. Dieser Prozess ist dann nötig, wenn die Architekturbeschreibung so unvollständig, falsch oder nicht vorhanden ist, sodass eine Bearbeitung der Software unmöglich wird. Es besteht auch eine Ähnlichkeit zum *program comprehension*, also dem Programmverstehen, das bewusst oder unbewusst von jedem Entwickler angewandt wird, der Änderungen an einer Software durchführt. Zu diesem Thema haben Maalej et al. [Maa+14] eine Studie erstellt, die 1477 Entwickler zur ihren Techniken und Strategien beim Programmverstehen befragt haben. Die Ergebnisse dieser Studie werden auch in dieser Arbeit verwendet und sollen die Architekturrekonstruktion beschleunigen, indem Irrwege und falsche Annahmen reduziert werden. Als Abgrenzung zum Programmverstehen soll bei der Architekturrekonstruktion als Resultat nicht nur das bessere Verständnis der Software stehen, sondern auch Artefakte, die die Architektur für nachfolgende Entwickler verständlich beschreiben. Deursen et al. [Deu02] haben bei einem Workshop zum Thema „*Software Architecture Recovery and Modelling*“ die Teilnehmer zu Trends und Techniken bei der Architekturrekonstruktion befragt und unter diesen Teilnehmern schlagen Riva et al. [Riv02] für die Architekturrekonstruktion ein inkrementelles, aus vier Phasen bestehendes Prozessmodell vor, auf das in dieser Arbeit aufgebaut werden soll.

2.4.4 Architekturanalyse

Eine Architekturanalyse stellt laut [BCK03] eine Untersuchung und Bewertung der Architektur einer Software dar. Anstatt Architekturanalyse wird auch häufig der Begriff der Architekturevaluation verwendet. In dieser Arbeit wird der Begriff der Architekturanalyse verwendet. Auch in der englischen Fachliteratur zu diesem Thema werden die Begriffe der *“architecture evaluation”* und *“architecture analysis”* verwendet. Bei der Architekturanalyse werden die Auswirkungen der Architektur auf die Qualitätsmerkmale der Software systematisch untersucht. Die Anzahl der beschriebenen Analyseverfahren für Softwarearchitekturen in der Fachliteratur ist sehr groß.

So gibt es keine standardisierten Verfahren, die für eine Analyse der Architektur empfohlen werden, jedoch viele Versuche, die vorhandenen Methoden zu klassifizieren. Babar et al. [BZJ04] untersuchen acht verschiedene Verfahren und definieren ein Framework um diese vergleichen zu können. Auch Dobrica et al.[DN02] vergleichen acht verschiedene Verfahren, die Menge der untersuchten Verfahren unterscheidet sich jedoch in zwei Verfahren. Alle diese Quellen vertreten die Ansicht, dass ein Ranking jedoch nicht sinnvoll ist und die Auswahl der Verfahren sich unterscheiden je nachdem, welche Ziele mit der Analyse verfolgt werden und welche Eigenschaften die untersuchte Software besitzt.

2.4.5 Metriken

Der IEEE Standard Glossar des Software Engineerings [RGK90] definiert Metrik als *"A quantitative measure of the degree to which a system, component, or process possesses a given attribute."*. Damit wird also eine quantitative Aussage über eine bestimmte Eigenschaft eines Systems oder eines Prozesses gemacht. Laut Ludewig et al. [RGK90] sind die Gründe für die Verwendung einer Metrik:

- Die Bewertung der Qualität von Produkten und Prozessen
- Das Quantifizieren von Erfahrungen
- Das Erstellen von Prognosen
- Das Unterstützen von Entscheidungen

Besonders bei der Bewertung von Architekturen sind Metriken nützlich, weil die Qualitätsmerkmale nicht ohne weiteres quantifiziert werden können. Erst durch die Wahl von geeigneten Metriken, können Architektureigenschaften quantifiziert und verglichen werden.

2.5 Spreadsheet Inspection Framework

Verschiedene Studien ([PLB08], [Pan98]) haben gezeigt, dass Spreadsheets, die im Unternehmensbereich eingesetzt werden, Fehler beinhalten und diese immer wieder zu Problemen und erhöhten Kosten führen können. Prüfwerkzeuge von Spreadsheets bieten bisher nur eine sehr eingeschränkte Funktionalität an oder sind nicht frei verfügbar. Deshalb wurde in der Abteilung Software Engineering des Instituts für Softwaretechnologie der Universität Stuttgart beschlossen ein neues Prüfwerkzeug unter einer Open-Source Lizenz zu entwickeln.

Das Spreadsheet Inspection Framework wurde ursprünglich von Sebastian Zitzelsberger [Zit12] entworfen, um die statische Prüfung von Spreadsheets zu ermöglichen. Es wurden exemplarisch drei statische Prüfmöglichkeiten umgesetzt und die spätere Erweiterung für dynamische Prüfungen wurde teilweise schon vorbereitet. Die Software wurde als Java-Framework implementiert und war betriebssystemunabhängig. Anschließend wurde die Software von Manuel Lemcke [Lem13] um dynamische Prüfungen, sogenannte Szenarien, erweitert. Zusätzlich wurde die Konfiguration der Prüfungen auf ein eigens dafür entwickeltes XML-Format umgestellt.

Als nächstes wurde die Software von Ehssan Doust [Dou13] in ein Socket-Client für Windows umgewandelt, damit das neu entwickelte Visualisierungswerkzeug mit Hilfe dieses Sockets kommunizieren konnte. Dieses Visualisierungswerkzeug wird Spreadsheet Inspection Framework Excel AddIn (SIFEI) genannt und wurde als Add-In für Excel in C# implementiert und ist somit nur unter Windows in Verbindung mit Microsoft Office nutzbar. Der in Java geschriebene Socket-Client wird nun als Spreadsheet Inspection Framework Service (SIFCore) bezeichnet.

In einem nächsten Schritt wurde von Jonas Scheurich [Sch14] die Konfiguration der Tests in das Visualisierungswerkzeug SIFEI integriert. Anschließend wurde SIFCore von Sebastian Beck [Bec14] um fünf weitere statische Prüfmöglichkeiten ergänzt. In der nächsten Iteration wurden dann SIFCore und SIFEI von Wolfgang Kraus [Kra14] erweitert, um die Plausibilität von Spreadsheet-Daten prüfen zu können. In der Folge wurde von Fabian Toth [Tot14] die automatisierte Prüfung und die verbesserte Darstellung von Fehlern in SIFEI umgesetzt. Zum Schluss wurde SIFEI noch einer weiteren Überarbeitung unterzogen, um die Benutzerfreundlichkeit und Robustheit zu verbessern.

2.6 Weitere Begriffsdefinitionen

Um den Lesefluss der Arbeit zu erhöhen, sollen hier noch einige weitere Begriffsdefinitionen aufgeführt werden, von denen der Autor der Meinung ist, dass sie dem Leser unklar sein könnten.

2.6.1 Entwickler

In dieser Arbeit wird häufig der Begriff des Entwicklers verwendet, dieser muss sich jedoch nicht auf eine einzelne Person beziehen, sondern kann auch eine Gruppe von Personen sein. Aus Gründen der Lesbarkeit wird in dieser Arbeit immer der Singular verwendet. Oft werden Entwickler, die sich mit speziellen Aufgaben beschäftigen gesondert benannt. So wird aus einem Entwickler, der sich mit der Softwarearchitektur beschäftigt, ein Architekt und ein Entwickler der Softwareprüfungen durchführt wird ein Tester. Auch hier verwendet diese Arbeit immer nur den Begriff Entwickler.

2.6.2 Artefakt

Ein Artefakt ist ein Produkt, das als Zwischen- oder Endergebnis in der Softwareentwicklung entsteht. Artefakte können zum Beispiel also Quellcode, Schaubilder oder Textdokumente sein. Auch Dinge wie Logdateien, Protokolle von Meetings oder E-Mails mit Fehlermeldungen von Benutzern sind Artefakte.

3 Architekturrekonstruktion

Bevor mit der Architekturanalyse begonnen werden kann, muss der Entwickler mit der Architektur vertraut sein. Das bedeutet sie soweit verstehen und erfassen zu können, dass er qualitative Aussagen über diese treffen kann. Wenn jedoch keine Architekturdokumentation vorhanden oder diese zum Beispiel aufgrund ungenügender Vollständigkeit oder Korrektheit nicht mehr nutzbar ist, muss in einem ersten Schritt die Dokumentation erstellt werden.

In diesem Kapitel wird ein Konzept entwickelt, das auf den in Kapitel 2 vorgestellten bisherigen Erkenntnissen im Bereich Architekturrekonstruktion und Programmverstehen basiert und mit dem die Architekturdokumentation in einem geordneten Prozess (wieder) hergestellt werden soll. Dieses Konzept wird am Beispiel des SIFCore angewendet und anschließend werden die bei der Durchführung gewonnenen Erkenntnisse vorgestellt. Der Grund für diese Architekturrekonstruktion von SIFCore ist, dass der Projektleiter eine *"hohe Erodierung"* der Architektur vermutet und gleichzeitig bemängelt, dass die Architekturdokumentation nicht mehr aktuell oder überhaupt nicht vorhanden ist.

3.1 Konzept

Da laut Standish et al. [Sta84] bei Softwareprojekten im Durchschnitt 50-80% der verwendeten Zeit für Wartung und 50-80% der Wartungszeit für das Programmverstehen verwendet werden, ist es wichtig, dass die Architekturrekonstruktion strukturiert und geplant wird. Ansonsten kann es zu großen Problemen kommen, da Risiko und Zeitaufwand nicht eingeschätzt werden können.

Für den Ablauf der Architekturrekonstruktion werden die folgenden Phasen definiert:

1. Identifikation der grundlegenden Komponenten und Konzepte
2. Inventarisierung der verfügbaren Quellen
3. Analyse der Quellen
4. Modellierung der Architektur
5. Erstellung der Architekturbeschreibung

Diese Phasen wurden ausgewählt unter der Berücksichtigung der vom IEEE vorgeschlagenen [Ieea] *best practices* und der Vorarbeit von Riva et al. [Riv02]. Die ersten beiden Phasen dienen als Vorbereitung und werden nur einmal ausgeführt. Die Phasen drei bis fünf werden dann in mehreren Iterationen durchlaufen. Mit diesem iterativen Vorgehen ist eine bessere Kontrolle

der benötigten Zeit möglich, wenn beispielsweise keine Erfahrungswerte für eine Aufwandsabschätzung vorliegen. Auch führt das iterative Vorgehen zu einer schrittweisen Verbesserung des Architekturmodells. Nach jeder Iteration können die Quellen spezifischer analysiert und dem Modell können weitere Abstraktionen hinzugefügt werden. Mit diesen Ergebnissen wird dann die Architekturbeschreibung verfeinert bis schließlich der gewünschte Detailgrad erreicht ist. Dieses iterative Vorgehen wurde von Rasool et al. [RA07] als auch von Riva et al. [Riv02] in ihren Publikationen vorgeschlagen.

3.1.1 Identifikation der grundlegenden Komponenten und Konzepte

In der ersten Phase werden die grundlegenden Komponenten und Konzepte der Softwarearchitektur identifiziert. Dazu gehört zum Beispiel die *Application Domain*, die festlegt, welche Probleme die Software lösen soll und welche Rolle ihr dabei zugeordnet wird. Diese Phase ist von der eingesetzten Technologie und den vorhandenen Artefakte abhängig und muss entsprechend durch den Entwickler angepasst werden. Aus diesem Grund ist eine detailliertere Definition dieser Phase nicht möglich. Der Entwickler sollte aber, bevor er mit der nächsten Phase beginnt, wissen mit welchen Technologien er arbeitet und welche Konzepte bei der Erstellung der Software verwendet und warum diese angewandt wurden. Der Entwickler sollte nach Abschluss dieser Phase Fragen wie die folgenden beantworten können:

- Um welche Art von Software handelt es sich?
- Wer benutzt die Software und warum?
- Aus welchen übergeordneten Komponenten besteht die Software?
- Welche Programmiersprache wird eingesetzt?
- Wird dabei eine bestimmte Sichtweise, wie zum Beispiel Objektorientierung, verwendet?
- Wie ist die Software in die Umgebung eingebettet und wie wird mit ihr interagiert?

Wichtig ist, dass in dieser Phase nicht bereits mit der detaillierten Analyse der Software begonnen und zu viel Zeit investiert wird. Diese Phase dient nur dazu, dem Entwickler einen grundlegenden Eindruck der Architektur zu vermitteln.

3.1.2 Inventarisierung der verfügbaren Quellen

In der zweiten Phase werden alle Quellen gesucht, die als Architekturbeschreibung dienen können. Dazu gehören selbstverständlich alle Dokumente, die zur Verwendung als Architekturbeschreibung gedacht sind, ebenso wie der Quellcode und die darin enthaltene Dokumentation. Aber auch Quellen, die nur indirekte Hinweise auf die Architektur geben, wie zum Beispiel die Beschreibung von Fehlern in einem Bugtracker oder die Nachrichten, die Entwickler als sogenannte „commit messages“ bei Änderungen schreiben, können nützlich sein. Als Quellen können also alle Artefakte dienen, die im Zuge der Entwicklung der Software entstanden sind.

3.1.3 Analyse der Quellen

In dieser Phase werden die in der vorherigen Phase gefundenen Quellen analysiert. Diese Phase ist dazu bestimmt, die für die Modellierung der Architektur notwendigen Informationen aus den Quellen zu extrahieren. Da im Normalfall die Informationsmenge sehr groß ist, wird von Riva et al. [Riv02] für diesen Entwicklungsabschnitt der Gebrauch von Werkzeugen empfohlen. Dieser ermöglicht es die großen Datenmengen, insbesondere den Quellcode, zu analysieren. Die Wahl der Werkzeuge muss dabei an die bei der Software verwendete Technologie sowie den Kenntnissen des Entwicklers angepasst werden. Der Einsatz von Werkzeugen kann auch zu schlechteren Ergebnissen führen als eine klassische Durchsicht des Quellcodes, insbesondere bei geringer Erfahrung der Entwickler mit dem Werkzeug.

Entscheidend ist auch, dass alle Artefakte auf ihre Korrektheit überprüft werden. Insbesondere Artefakte, die nicht Quellcode sind, wie textuelle Dokumentation oder Komponentendiagramme, die höhere Abstraktionsebenen beschreiben, müssen anhand des Quellcodes überprüft werden. Da wahrscheinlich kein Artefakt fehlerfrei sein wird und der Aufwand für eine Überprüfung beliebig groß sein kann, muss der Entwickler selbst abschätzen können wie fehlerfrei ein Artefakt ist und wie viel Zeit für eine Überprüfung investiert werden soll. Die Analyse der Quellen bedeutet auch immer ein Abwägen der aufgewendeten Zeit und der Qualität der Ergebnisse.

3.1.4 Modellierung der Architektur

In dieser Phase werden die vorhandenen einzelnen Informationen, die aus den Quellen extrahiert wurden, in einen Zusammenhang gesetzt. Ziel ist es, eine umfassende Sicht (*"high level view"*) der Komponenten mit einem hohen Abstraktionsniveau zu erstellen, sodass die Sicht nur die für die Architektur relevante Details enthält. Ausgehend von den in der ersten Phase identifizierten grundlegenden Komponenten und Konzepten, wird die Architektur stückweise, mit den in der zweiten Phase gefundenen und in Phase drei analysierten Informationen, erweitert. Wenn alle bisher analysierten Ergebnisse genügend abstrahiert und in das Modell integriert wurden, kann entweder, wenn das Modell einen zufriedenstellenden Zustand erreicht hat, die nächste Phase begonnen oder die noch fehlenden Informationen in einer weiteren Analysephase erarbeitet und anschließend integriert werden.

3.1.5 Erstellung der Architekturbeschreibung

In einer letzten Phase wird jetzt die Architekturbeschreibung erstellt beziehungsweise vervollständigt. Ohne die Architekturbeschreibung ist es nicht möglich, den entstandenen Entwurf effektiv zu kommunizieren oder zu beschreiben. Es droht der Verlust von bereits geleisteter Arbeit, wenn die Architekturbeschreibung nicht die in den vorherigen Phasen erarbeiteten Ergebnisse darstellen kann. Eine Möglichkeit, eine Architektur und die damit verbundenen verschiedenen Sichten (im Kapitel 2 vorgestellt) zu beschreiben, bietet das 4+1 Sichtenmodell von Kruchten et al. [Kru95]. Dieses Modell kann dann mit einer Architecture Description Language (ADL)

semi-formal beschrieben werden. In dieser Arbeit wird die Unified Modeling Language (UML)[1] verwendet, um die verschiedenen Sichten textuell und visuell zu beschreiben. Dabei wird das in Tabelle 3.1 definierte Mapping verwendet. Der Grund für diese Entscheidung ist die hohe Verbreitung von UML als ADL. UML wird auch in der Lehre der Universität Stuttgart großflächig verwendet und somit ist sichergestellt, dass auch nachfolgende Entwickler von SIFCore diese Architekturbeschreibung verstehen und bearbeiten können.

Sicht	verwendetes UML Diagramm
Logical view	Klassen-, Kommunikations- oder Sequenzdiagramm
Development view	Komponentendiagramm
Process view	Aktivitätsdiagramm
Physical view	Verteilungsdiagramm
Scenarios	Use-Case-Diagramm

Tabelle 3.1: Zuordnung der Architektursichten zu UML-Diagrammen

3.2 Umsetzung

Nach der Ausarbeitung des Konzeptes wird die Architekturbeschreibung von SIFCore im nächsten Schritt nun rekonstruiert.

3.2.1 Identifikation der grundlegenden Komponenten und Konzepte

Für die erste Phase der Architekturrekonstruktion, die hauptsächlich als Vorbereitung und Orientierung dient, wird eine Quelle benötigt, die nach Möglichkeit diese Informationen in kompakter und verständlicher Form liefert. Bei kommerziellen Softwareprojekten ist dies (falls vorhanden) das Handbuch für Entwickler, im Falle von SIF stehen die Abschlussarbeiten der bisherigen Entwickler zur Verfügung. Eine textuelle Beschreibung der Architektur und der grundlegenden Komponenten von SIFCore findet sich in der Arbeit von [Zit12] in Kapitel 8.2. Diese Arbeit stellte die erste Iteration der Entwicklung von SIF dar und aus diesem Grund ist die Beschreibung der Architektur nicht vollständig. Da die aktuelleren Arbeiten jedoch keine allgemeine Architekturbeschreibung bieten, muss auf diese erste Beschreibung zurückgegriffen werden und die Änderungen der nachfolgenden Arbeiten müssen im Zuge der nächsten Rekonstruktionsphasen überprüft und integriert werden. Ausgehend von dieser Basis können die folgenden Konzepte von SIF wie folgt identifiziert werden:

- SIFCore ist in Java geschrieben und folgt dem Programmierparadigma der Objektorientierung.

[1] http://www.omg.org/spec/UML/

- SIF war als Java-Framework konzipiert und sollte in andere Java-Programme integriert werden. Allerdings wurde dieses Konzept in einer der späteren Iterationen geändert und SIF wurde in eine *standalone* Java-Applikation (ab jetzt SIFCore genannt) umgewandelt.
- Die Funktionsweise von SIFCore folgt der Metapher der *technischen Inspektions-Werkstatt*.
- SIFCore soll verschiedenartige Prüfungen von Spreadsheets ermöglichen um Fehler zu erkennen oder aber vor potenziellen Fehlern zu warnen.
- SIFCore bietet dazu die Möglichkeit sogenannte *Inspektions-Aufträge* zu konfigurieren.
- SIFCore kann diese *Inspektions-Aufträge* auf eingelesene Spreadsheets anwenden und diese so prüfen.
- SIFCore kann die Ergebnisse von Prüfungen in Form von Berichten exportieren.

Des weiteren können die folgenden Komponenten von SIFCore identifiziert werden:

- Eine Komponente zur Programmflusskontrolle (genannt *FrontOffice*)
- Eine Komponente zur Speicherung von Spreadsheet-Inhalten (genannt *Model*)
- Eine Komponente zur Verwaltung von Prüfungen (genannt *TechnicalDepartment*)
- Eine Komponente zum Importieren von Spreadsheets in die Datenhaltungskomponente und Exportieren von Prüfergebnissen (genannt *IO*)

Damit ist die Identifikation der grundlegenden Komponenten und Konzepte von SIFCore abgeschlossen und es kann mit der nächsten Phase fortgefahren werden.

3.2.2 Inventarisierung der verfügbaren Quellen

SIF besteht aus zwei verschiedenen Projektarchiven. Zum einen gibt es das Projektarchiv von SIFCore, das die Java-Applikation beinhaltet und zum anderen gibt es das Projektarchiv von SIFEI, das den Visualisierungsclient als Excel AddIn in C# beinhaltet. In diesen Projektarchiven sind alle Artefakte gespeichert und beinhalten, mit Ausnahme der Berichte der Abschlussarbeiten, alle verfügbaren Quellen. Weil sich diese Architekturrekonstruktion nur auf SIFCore bezieht, werden alle Artefakte im SIFEI-Projektarchiv vorerst ignoriert. Als Quellen für die weitere Analyse werden in Betracht gezogen:

- Der gesamte Java-Quellcode von SIFCore inklusive Unittests und Testdaten
- Die Berichte der Abschlussarbeiten von Sebastian Zitzelsberger [Zit12], Manuel Lemcke [Lem13], Ehssan Doust [Dou13], Jonas Scheurich[Sch14], Sebastian Beck [Bec14], Wolfgang Kraus [Kra14] und Fabian Toth [Tot14]
- Die *commit messages* der Entwickler in den beiden GitHub-Projektarchiven
- Die Beschreibungen und Kommentare der als *Issues* bezeichneten Fehlermeldungen und Feature-Wünsche auf GitHub

In den Abschlussarbeiten finden sich einige Artefakte, die als Architekturbeschreibung dienen können. Weil sich die Architektur während der Entwicklung verändert, beziehen sich einige Artefakte womöglich auf ältere Versionen der Architektur. Daher werden besonders ältere Artefakte bei der späteren Analyse auf ihre Relevanz hin überprüft. Da sich aber viele, für eine Architekturbeschreibung notwendige, Artefakte nicht mehr in späteren Abschlussarbeiten finden, werden einige Artefakte aus den früheren Arbeiten benötigt, obwohl diese nicht die aktuelle Architektur beschreiben. Die in Tabelle 3.2 aufgelisteten Artefakte werden deshalb (in chronologischer Reihenfolge) als Quellen für die Analyse ausgewählt.

Name des Artefakts	Quelldokument
Abbildung 6.1.: SIF als technische Grundlage [...]	[Zit12]
Abbildung 6.2.: Ablauf einer Inspektion [...]	[Zit12]
Abbildung 8.1.: Architektur von SIF	[Zit12]
Abbildung 8.2.: Darstellung des Spreadsheet-Inventars	[Zit12]
Abbildung 8.3.: Die Komponenten von SIF im Detail	[Zit12]
Abbildung 2.1.: Verwendung von SIF durch eine Anwendung	[Lem13]
Abbildung 3.2.: Ablauf in der zweiten Ausbaustufe	[Lem13]
Abbildung 3.3.: Inspektion dynamischer Vorschriften	[Lem13]
Abbildung 4.1.: Aufbau von SIF	[Lem13]
Abbildung 4.2.: Aufbau einer dynamischen Vorschrift	[Lem13]
Abbildung 4.3.: Vererbungshierarchie der Bedingungen	[Lem13]
Abbildung 4.4.: Vererbungshierarchie der Prüfstände	[Lem13]
Abbildung 3.2.: Zeitlicher Ablauf der Prüfung [...]	[Sch14]
Abbildung 5.2.: Architektur der Komponenten SIFEI und SIF	[Sch14]

Tabelle 3.2: Artefakte zur weiteren Analyse

3.2.3 Analyse der Quellen & Modellierung der Architektur

Eine Beschreibung aller Arbeitsschritte während der Analyse und der Modellierung erweist sich als unpraktikabel, weil diese Phasen stark von einander abhängen. Die Umsetzung aller Iterationen wird deshalb in einem einzigen Schritt dokumentiert.

In dieser Arbeit werden die Entwicklungsumgebung IntelliJ IDEA[2] und das Analysetool JArchitect[3] als Werkzeuge zur Unterstützung der Analyse verwendet. Beide Werkzeuge ermöglichen es verschiedene Abstraktionen automatisiert zu erstellen. So zeigt die Analyse der Quellen, dass zwar einige Entwickler für Teile der Architektur Beschreibungen zur Verfügung gestellt haben (zum Beispiel Abbildungen 4.3. und 4.4 in der Arbeit von [Lem13]), aber für andere Teile der Architektur fehlen diese Informationen. Mit Hilfe der genannten Werkzeuge können die fehlenden Beschreibungen auf Basis des Quellcodes generiert werden.

[2] https://www.jetbrains.com/idea/
[3] http://www.jarchitect.com/

Neben der Zeitersparnis haben diese Werkzeuge den Vorteil, dass die erzeugten Artefakte keine formalen Fehler enthalten, im Gegensatz zu Diagrammen die manuell erstellt werden. Ein Nachteil der Werkzeuge ist allerdings, dass sie nur in einem sehr begrenzten Umfang zur Abstraktion beitragen. Sie können keine *virtuellen* Komponenten erkennen, die zwar vom Entwickler erdacht sind, aber keine direkte Umsetzung im Quellcode haben. Diese sind auch abhängig von der jeweiligen Programmiersprache. Im Falle von SIFCore werden nur die Java-Elemente *package* und *class* als Komponenten erkannt. Auch sind die meisten erstellten Diagramme zu detailliert. Das Ziel der Modellierungsphase ist die Abstraktion und damit die Reduzierung auf das Wesentliche. Mit einer manuellen Nachbearbeitung, etwa durch Löschen von unwichtigen oder dem Hinzufügen von übergeordneten Komponenten, kann die erforderliche Abstraktionsebene erstellt werden die vom Werkzeug nicht erreicht wird.

3.2.4 Erstellung der Architekturbeschreibung

Da die Architektur von SIFCore im Zuge dieser Arbeit verändert wird und dementsprechend die Architekturbeschreibung auch nach der Bearbeitung aktualisiert werden muss, muss die Beschreibung der alten Architektur nur so vollständig sein, dass eine Analyse und eine Überarbeitung möglich ist. Aus diesem Grund wird auch auf die Darstellung der *"Physical view"* Sicht und der *"Process view"* Sicht verzichtet. Die folgenden Artefakte beschreiben dabei die Komponenten und Sichten der Architektur von SIFCore.

- Komponentendiagramm mit den in SIFCore enthaltenen Komponenten der oberen Abstraktionsebenen und ihren Abhängigkeiten zu anderen Komponenten
- Klassendiagramme der Komponenten FrontOffice, Model, TechnicalDepartment und IO
- Programmablaufplan (nach DIN 66001) einer Spreadsheet-Inspektion
- Sequenzdiagramm des Spreadsheet-Imports
- Sequenzdiagramm (UML2) der Analyse von Formeln beim Spreadsheet-Import
- Sequenzdiagramm (UML2) der dynamischen Prüfung von Spreadsheets
- Use-Case-Diagramm der dynamischen Prüfung von Spreadsheets

Abbildung 3.1 zeigt das Komponentendiagramm mit den nun identifizierten Komponenten der obersten Abstraktionsebene. Damit zeigt es die *Development view* genannte Sicht und beschreibt das System aus dem Blickwinkel der Entwickler. Das Diagramm zeigt die einzelnen Komponenten, die jeweils als Java-Package implementiert sind. Die Verbindungen der Komponenten sind dabei entweder Abhängigkeitsbeziehungen oder Assoziationen. Der Unterschied besteht darin, dass von Assoziationen Instanzen angelegt werden können, wenn also zwischen zwei Klassen eine Assoziation besteht, dann sind auch zwei Instanzen der Klassen über eine Objektbeziehung miteinander verbunden. In diesem Diagramm werden nur Abhängigkeitsbeziehungen vom Typ *«create»* verwendet, die durch eine gestrichelte Linie mit einem Pfeil dargestellt werden. Diese Abhängigkeitsbeziehung bedeutet, dass die Quell-Komponente Objekte erzeugt, die nicht der eigenen, sondern der Ziel-Komponente zugeordnet werden.

Assoziationen werden mit einer durchgehende Linie dargestellt. Meistens werden Assoziationen durch das Senden einer Nachricht oder den Aufruf einer Methode umgesetzt. Die hier verwendeten Assoziationen, sind alles Kompositionen und durch eine ausgefüllte Raute gekennzeichnet. Die Komposition zeigt an, dass die Quelle verantwortlich für die Erstellung und Speicherung des Ziels ist. Folglich kann das Ziel der Komposition nicht ohne die Quelle existieren.

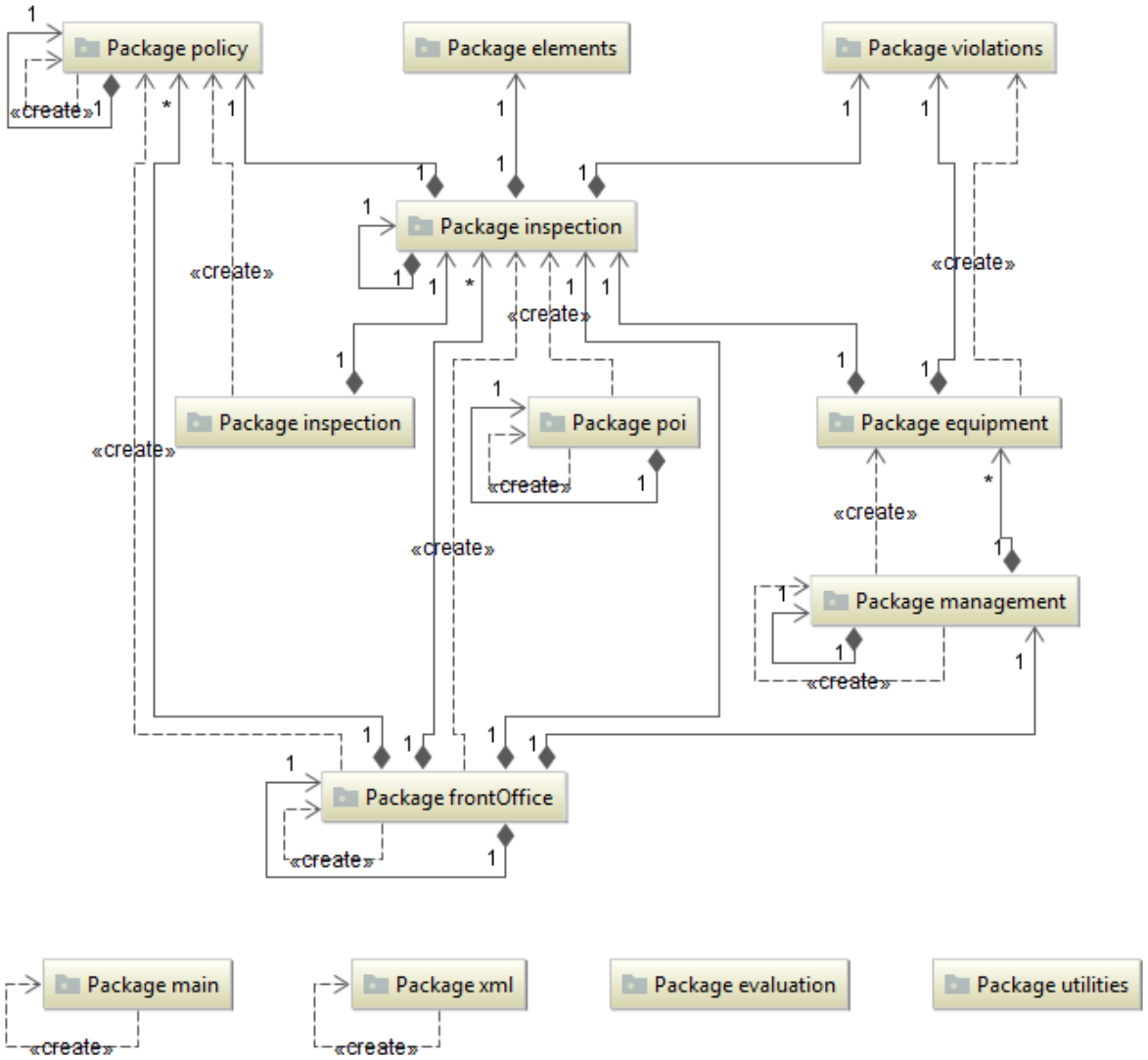

Abbildung 3.1: Komponenten von SIFCore vor der Überarbeitung

Der Umfang der restlichen Dokumente ist teilweise sehr groß, deshalb wird auf eine Darstellung in dieser Arbeit verzichtet. Alle hier genannten Dokumente finden sich im SIFCore-Projektarchiv auf der beigelegten CD.

Auf eine weitere Beschreibung der Architektur wird verzichtet, da eine Überarbeitung mit der erstellten Dokumentation bereits möglich ist. Die Architektur wird sich im Rahmen des Reengi-

neering stark verändern und somit wird auch eine aktualisierte Beschreibung der Architektur benötigt. Eine differenziertere Dokumentation der bisherigen Architektur erscheint deshalb nicht sinnvoll.

3.3 Ergebnisse

Trotz eingehender Analyse aller verfügbaren Artefakte sowie der intensiven Beschäftigung mit dem Thema „*architecture reconstruction*“, war die Umsetzung der Architekturrekonstruktion von verschiedenen Problemen geprägt.

Weil die Quellen mit hoher Abstraktionsebene in den oben beschriebenen Fällen nicht korrekt oder informativ genug waren, musste die Architektur aus dem Quellcode extrahiert und mit Hilfe eigener Abstraktionsregeln erstellt werden. Diese Arbeit war zwar mühsam und zeitaufwendig, führte aber im Gegenzug auch zu einem umfangreichen und detaillierten Verständnis des Quellcodes.

Als nachteilig erwies sich, dass die Ergebnisse einer Arbeit nicht vollständig in den dafür vorgesehenen Projektarchiven abgelegt wurden, sondern dass insbesondere Artefakte, die höhere Abstraktionsebenen beschreiben wie Klassen-, Struktur- oder Ablaufdiagramme, mangelhaft oder gar nicht vorhanden waren. Diese Artefakte könnten durchaus im Verlauf der Arbeiten entstanden sein, gingen aber im Laufe der Entwicklung wieder verloren. Auch ist es ungünstig, dass Metainformationen der einzelnen *commits* zu den Versionsverwaltungssystemen, die die Entwickler durchgeführt haben, verloren gegangen sind weil die Ergebnisse einer Arbeit zu einem einzigen (anonymen) *commit* zusammengefasst wurden. Trotz dieser Umstände konnte die Architektur zur Zufriedenheit des Autors rekonstruiert und damit auch dokumentiert werden.

4 Architekturanalyse

Die Architekturanalyse soll es ermöglichen, die Architektur und damit die Qualität einer Software zu bewerten. So sollen Aussagen hinsichtlich der Erfüllung bestimmter Qualitätsmerkmale getroffen oder auch verschiedene Architekturen miteinander verglichen werden. In diesem Kapitel wird die Architektur von SIFCore untersucht und bewertet, um mögliche Schwächen zu identifizieren, damit sie in einem nächsten Schritt verbessert werden können. Hierfür werden, im Rahmen einer Vorauswahl, einige Verfahren bestimmt, die für eine solche Bewertung geeignet erscheinen. Anschließend werden diese Verfahren durchgeführt und die Ergebnisse vorgestellt, um am Ende dieses Kapitels eine konkrete Bewertung der Architektur vorzunehmen.

4.1 Vorauswahl der Verfahren

Wie in Kapitel 2 bereits genannt, gibt es zahlreiche Auffassungen, mit welchen Methoden und auf welcher Ebene eine Bewertung der Architektur möglich und sinnvoll ist. Abowd et al. [Abo+97] haben die Teilnehmer eines Workshops mit dem Titel *„Industrial practice of software architecture evaluation“* befragt, welche Verfahren zur Bewertung von Architekturen in ihren Unternehmen genutzt werden. Dabei unterscheiden sie fünf Kategorien zur Architekturanalyse: *Fragenkataloge*, *Checklisten*, *Szenarien*, *Metriken* und *Simulationen und Experimente*. Diese Kategorisierung stellt die Grundlage für die Auswahl von Bewertungsverfahren in dieser Arbeit dar. Es ist wichtig, Verfahren aus möglichst allen Kategorien zu verwenden, um eine hohe Diversität der Verfahren zu gewährleisten.

4.1.1 Fragenkataloge

Die Verwendung eines *Fragenkatalogs*, das heißt das Befragen aller Projektbeteiligten zum Zustand der Architektur mittels eines zuvor zusammengestellten Katalogs, stellt sich im Rahmen dieser Vorauswahl als ungeeignet für die Bewertung von SIFCore heraus. Gründe hierfür sind, dass die Anzahl der für eine Befragung zur Verfügung stehenden Personen sehr gering und die Ausarbeitung eines geeigneten Fragebogens zu zeitaufwendig ist. Eine oberflächliche Analyse, mit nur einer Person und nicht ausreichend vorbereiteten Fragen, würde nur zu unbefriedigenden Ergebnissen führen und unter diesen Gesichtspunkten wird auf eine entsprechende Analyse in dieser Arbeit verzichtet.

4.1.2 Checklisten

Bei der Verwendung einer *Checkliste* wird im Rahmen eines Reviews überprüft, in wieweit die Architektur die in der Checkliste aufgelisteten Punkte erfüllt. Die Auswahl der vorhandenen Punkte, die bei diesem Verfahren geprüft werden, ist dabei offensichtlich wichtig für den Erfolg und die Aussagekraft des Verfahrens. Diese Checkliste wird innerhalb einer Organisation über mehrere Projekte hinweg erstellt und eingesetzte Werkzeuge und Programmiersprachen werden dabei berücksichtigt. Im Falle von SIFCore ist jedoch keine solche Checkliste vorhanden. Ihre Erstellung hätte sehr viel Zeit benötigt, daher wurde auf eine entsprechende Analyse in dieser Arbeit verzichtet. Als Ersatz für eine umfangreiche Checkliste wurde stattdessen, im Rahmen einer Durchsicht, die Einhaltung der funktionalen und nichtfunktionalen Bedingungen in einem nicht-formal definiertem Verfahren überprüft.

4.1.3 Szenarien

Verfahren in der Kategorie *Szenarien* benötigen keine Implementierung der Architektur und eignen sich deshalb für eine frühe Analyse. In dieser Arbeit ist die Implementierung der Software zwar schon vorhanden, aber weil sich, die in Kapitel 1 formulierten Fragen auf alle verfügbaren Verfahren beziehen, sollen nicht ausschließlich Verfahren eingesetzt werden, die eine Implementierung voraussetzen. In Abschnitt 2.4.4 werden einige Arbeiten vorgestellt, die auf Szenarien basierende Analyseverfahren vergleichen. Diese Verfahren beziehen sich auf bestimmte Qualitätsmerkmale der Architektur. Deshalb muss vor Auswahl der entsprechenden Verfahren festgelegt werden, welche Qualitätsmerkmale der Architektur untersucht werden sollen.

SIFCore wurde in den bisherigen Entwicklungsphasen von verschiedenen Entwicklern betreut und wird auch nach diesem Reengineering in weiteren Entwicklungsphasen von weiteren Entwicklern verändert werden. Der Projektleiter vermutet, dass die große Anzahl an beteiligten Entwicklern für die Erodierung der Architektur mitverantwortlich ist. Aus diesem Grund erscheint es sinnvoll, als zentrales Qualitätsmerkmal die *Modifizierbarkeit* (engl. modifiability) zu untersuchen.

4.1.4 Metriken

Für Verfahren dieser Kategorie werden verschiedene Metriken der Software (hauptsächlich mit Hilfe von Werkzeugen) bestimmt. Aus den Werten der Metriken werden dann Rückschlüsse auf die Architektur gezogen. Dabei existiert keine direkte Korrelation zwischen Metriken und Qualitätsmerkmalen der Software, das heißt der Architektur. Die Metriken müssen vorab noch interpretiert werden und können meist nur in Kombination mit anderen Metriken Hinweise auf die Erfüllung eines bestimmten Qualitätsmerkmals geben.

Es gilt bei der Verwendung von Metriken, die von Bowers et al. [Bou13] aufgezeigten vier häufigsten Fehler zu vermeiden, die bei der Untersuchung von 400 Softwareprojekten identifizieren werden konnten.

Als Erstes müssen Metriken immer hinsichtlich der Architektur und damit eines Qualitätsmerkmals interpretiert werden. Metriken ohne festgelegte Bedeutung für die Architektur sind demzufolge nutzlos.

Als Zweites muss die Bedeutung einer Veränderung des Werts einer Metrik in den korrekten Kontext gesetzt werden. Veränderungen des Wertes können auch anderen Umständen geschuldet sein, als einer Veränderung des mit dieser Metrik assoziierten Qualitätsmerkmals.

Als Drittes sollte der Fokus auf eine einzelne Metrik vermieden werden. Wenn ein Qualitätsmerkmal nur anhand von einzelnen beziehungsweise zu wenigen Metriken gemessen wird, dann wird das Qualitätsmerkmal unter Umständen in unzulässiger Weise vereinfacht und die Aussage verfälscht.

Als Letztes sollte auch vermieden werden, zu viele Metriken messen und beeinflussen zu wollen. Dies führt zu einem erhöhten Arbeitsaufwand und kann zu widersprüchlichen Ergebnissen führen. So kann der Wert einer Metrik verbessert werden, zeitgleich verschlechtern sich möglicherweise aber Werte von anderen Metriken. Bei der Verbesserung der Werte dieser Metriken verschlechtert sich wiederum der Wert der ersten Metrik.

In dieser Arbeit werden einige der von den Werkzeugen angebotenen Metriken ausgewählt und diese dann genauer betrachtet.

4.1.5 Simulationen und Experimente

Eines der geforderten Ziele dieser Abschlussarbeit ist die Auswertung des öffentlichen „Issue-Trackers“, die deshalb als eigenständiges Verfahren zur Architekturanalyse in dieser Arbeit verwendet wird. Weil das Überprüfen der gemeldeten Issues ein Ausführen der Software nötig macht, wird dieses Verfahren der Kategorie *Simulationen und Experimente* zugeordnet.

4.1.6 Ausgewählte Verfahren

Die folgenden Verfahren werden ausgewählt, um die Architektur von SIFCore zu analysieren:

1. Die szenariobasierten Analyseverfahren SAAM und ALMA. SAAM, weil es als Erstes veröffentlicht wurde und bereits einige Fallstudien mit praktischen Erfahrungen vorliegen. ALMA, weil sich das Verfahren auf die Modifizierbarkeit der Architektur konzentriert, welches als zentrales Qualitätsmerkmal zur Untersuchung der Architektur bestimmt wurde.
2. Eine Untersuchung verschiedener Metriken mit Hilfe der Werkzeuge IntelliJ IDEA[1], Teamscale[2] und Sonarqube[3]. Dabei sollen sowohl einzelne Metriken untersucht werden, die sich auf ausgewählte Merkmale der Architektur beziehen, als auch Pseudometriken verwendet

[1] https://www.jetbrains.com/idea/
[2] https://www.cqse.eu/de/produkte/teamscale/ueberblick/
[3] https://www.sonarqube.org/

werden, die eine umfassende Bewertung der Architektur ermöglichen. Außerdem wird die Metrik DSM eingesetzt, da diese sich explizit auf die Softwarearchitektur bezieht.

3. Die Auswertung des öffentlichen Issue-Trackers, um bereits in früheren Entwicklungsabschnitten gefundene Schwächen und Probleme der Architektur von SIFCore einzuordnen.
4. Eine abschließende Durchsicht der Architektur, um eventuell mit den bisherigen Verfahren nicht abgedeckte Probleme zu finden.

Mit dieser Auswahl wird ein breites Spektrum an verfügbaren Analyseverfahren abgedeckt. Es werden sowohl Verfahren genutzt, die zu einem frühen Zeitpunkt in der Softwareentwicklung eingesetzt werden können, wie SAAM oder ALMA. Ebenso kommen Verfahren zum Einsatz, die eine Implementierung der Architektur benötigen, wie die auf Metriken basierende Analyse oder Simulationen und Experimente.

4.2 Szenarien

Da alle szenariobasierten Analyseverfahren sogenannte Szenarien benötigen und ein Vergleich der Verfahren möglich sein soll, werden für beide angewendeten Verfahren die gleichen Szenarien verwendet. Die Definition des Begriffs *„Szenario“* wird sowohl von Kazman et al. [Kaz+96] für SAAM, als auch von Lassing et al. [Las02] für ALMA definiert. Weil sich die Definitionen weitestgehend überschneiden, ist die Definition des Begriffs „Szenarios“ soweit unstrittig und die Verwendung von denselben Szenarien für beide Verfahren möglich.

Szenarien beschreiben die Qualitätsanforderungen an eine Architektur mittels Auswahl einer konkret beschriebenen Aufgabe. Diese Aufgabe kann sich auf die aktuelle Architektur und die dort vorhandenen Operationen beziehen und wird dann als *operational scenario* bei ALMA bezeichnet. Oder die Aufgabe bezieht sich auf die zukünftige Verwendung des Systems und wie die Architektur durch diese zukünftige Verwendung beeinflusst wird, um dann bei ALMA als *evolutionary scenario* und bei SAAM als *indirect scenario* bezeichnet zu werden.

Diese evolutionären Szenarien verwenden das Prinzip, das von Parnas et al. [Par72] für die Änderung von Softwaresystemen beschrieben wurde: Es wird geprüft, unter welchen Umständen getroffene Entwurfsentscheidungen verändert werden müssen und wie sich diese Veränderungen auf das Gesamtsystem auswirken. Da in dieser Arbeit nur das Qualitätsmerkmal *Modifizierbarkeit* untersucht wird, werden alle Szenarien als evolutionäre, beziehungsweise indirekte, Szenarien umgesetzt.

4.2.1 Verfahren

Im folgenden Abschnitt werden die einzelnen Aktivitäten und der zugrundeliegende Prozess der beiden angewendeten Verfahren vorgestellt.

SAAM - Software architecture analysis method

Die SAAM wird als das erste Verfahren zur Analyse der Architektur angesehen, das auch veröffentlicht wurde [Kaz+94]. Ursprünglich war SAAM dazu gedacht, verschiedene konkurrierende Architekturen zu vergleichen, bevor diese implementiert werden müssen. SAAM wurde aber inzwischen auch für die Bewertung von einzelnen Architekturen angepasst [Kaz+96] und dient als Grundlage für viele andere auf Szenarien basierende Analyseverfahren. Kazman et al. [Kaz+96] haben in ihrem technischen Bericht, in dem einige Architekturanalysen bei realen Softwaresystemen untersucht wurden, die folgenden Aktivitäten von SAAM definiert:

- **Beschreibung der Architektur**

  Die Architektur des untersuchten Systems sollte in einer Art und Weise beschrieben werden, die alle bei der Analyse beteiligten Personen verstehen können. Diese Beschreibung sollte alle Komponenten und ihre Beziehungen untereinander erfassen.

- **Entwicklung der Szenarien**

  Wichtig ist, dass alle Anwendungsfälle, die das System unterstützen soll durch Szenarien abgedeckt werden. Es entstehen also Szenarien, die für verschiedene Rollen (zum Beispiel Kunde, Systemadministrator oder Entwickler) des Systems relevant sind.

- **Kategorisierung und Priorisierung der Szenarien**

  Es werden die entwickelten Szenarien nach ihrer Art und Wichtigkeit sortiert. So soll sichergestellt werden, dass die gewünschten Architekturmerkmale durch die Szenarien abgedeckt sind.

- **Evaluation der Szenarien**

  Für jedes indirekte Szenario wird eine Liste mit an der Architektur nötigen Änderungen erstellt und die Kosten für diese Änderung geschätzt. Als Änderungen gelten dabei das Hinzufügen, Löschen oder Bearbeiten einer Komponente oder einer Verbindung zwischen Komponenten. Es wird anschließend eine Zusammenfassung der Ergebnisse erstellt.

- **Interaktionen aufdecken**

  Verschiedene indirekte Szenarien müssen unter Umständen die gleichen Komponenten des Systems verändern. Ein solcher Konflikt wird als Interaktion zwischen den Szenarien bezeichnet. SAAM bevorzugt Architekturen, bei denen möglichst wenige Interaktionen auftreten.

- **Abschließende Bewertung**

  Zum Schluss wird die Architektur unter Berücksichtigung der Szenarien und Interaktionen von allen Projektbeteiligten bewertet.

Die Aktivitäten werden dabei nicht einfach der Reihe nach ausgeführt, sondern besitzen Abhängigkeiten, die als Prozessmodell in Abbildung 4.1 dargestellt sind. Abgesehen davon wird der Prozess nicht weiter spezifiziert.

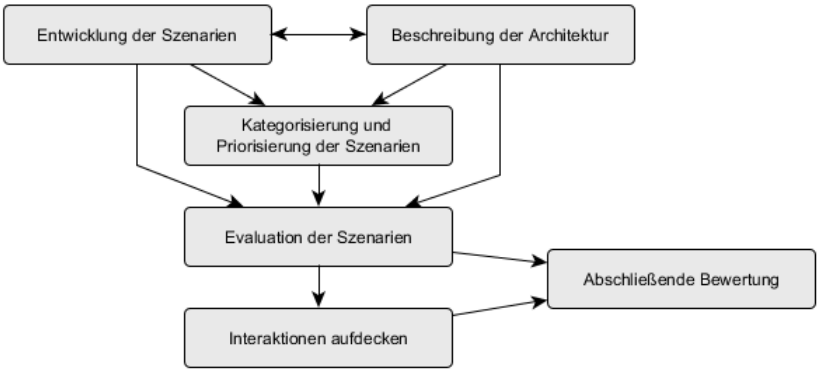

Abbildung 4.1: SAAM-Prozess aus [Kaz+96]

ALMA - Architecture Level Modifiability Analysis

ALMA ist wie SAAM ein auf Szenarien basierendes Analyseverfahren. Im Gegensatz zu SAAM ist ALMA aber von Anfang darauf ausgerichtet, nur solche Szenarien zu untersuchen, in denen die Architektur verändert wird. Dazu definiert das Verfahren einen Prozess, um die Architektur hinsichtlich des Qualitätsmerkmals *Modifizierbarkeit* zu bewerten. Von Lassing et al. [Las02] werden die folgenden Aktivitäten beschrieben:

- **Ziele setzen**

  Bevor die eigentliche Analyse gestartet wird, müssen die Ziele, die mit der Analyse erreicht werden sollen, benannt werden. Das Definieren einer eigenen Aktivität für die explizite Nennung der Ziele hebt die Bedeutung hervor, die diese im Prozess haben. Normalerweise ist das Ziel von ALMA eine Risikoanalyse, eine Kostenanalyse oder ein Vergleich von Architekturen.

- **Beschreibung der Architektur**

  Die Architektur des untersuchten Systems muss so beschrieben sein, dass im Zuge von Modifikationen an der Architektur, Vorhersagen über nötige Änderungen an den Komponenten getroffen werden können.

- **Szenarien auswählen**

  Bei dieser Aktivität werden die für die Analyse relevanten Szenarien ausgewählt und ausgearbeitet.

- **Evaluation der Szenarien**

  Bei der Evaluation der Szenarien werden die benötigten Änderungen durch eine Analyse der Auswirkungen des Szenarios auf dem *„architecture-level“* bestimmt. Eine solche Analyse besteht laut Bengtsson et al. aus den drei Schritten:

  1. Identifizierung der unmittelbar betroffenen Komponenten.
  2. Die Auswirkungen der Änderungen auf diese Komponenten untersuchen.
  3. Mögliche Seiteneffekte erkennen, die durch die geplanten Änderungen verursacht werden könnten.

- **Abschließende Bewertung**

  Zum Schluss werden die bei der Evaluation gewonnenen Erkenntnisse interpretiert, um die Auswirkungen auf die Architektur zu bestimmen. Die Interpretation der Ergebnisse ist dabei abhängig vom Ziel der Analyse und den Anforderungen an die Architektur.

In Abbildung 4.2 wird der Prozess von ALMA wie er von Bengtsson et al. [Ben+] beschrieben wird, dargestellt.

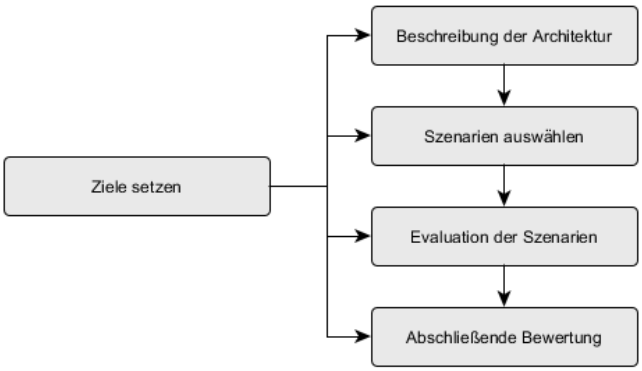

Abbildung 4.2: ALMA-Prozess aus [BG]

4.2.2 Ausgewählte Szenarien

ALMA und SAAM basieren beide auf der Evaluation von Szenarien. Um die Verfahren besser vergleichen zu können, werden für Beide die gleichen Szenarien verwendet. Dies ist auch möglich, weil der Prozess zur Auswahl der Szenarien in beiden Verfahren so frei definiert ist, dass er übereinstimmt.

Szenario 1: Erstellen einer neuen Prüfung

Es soll eine neue Prüfung für Spreadsheets erstellt werden. Bei der Prüfung sollen Inhalte von Spreadsheets (und damit Inhalte von einzelnen Zellen) sowie deren Metadaten (zum Beispiel die Anzahl an Verweisen von anderen Zellen auf diese Zelle) verwendet werden. Es soll zudem möglich sein, Inhalte von Zellen zu ändern, um auf dem Spreadsheet vorhandene Funktionen auszuführen. Die Prüfung soll außerdem einige Konfigurationsparameter beinhalten, die durch den User geändert werden können. Die Verstöße, die bei diesen Prüfungen auftreten, sollen in den Prüfreport aufgenommen werden.

Szenario 2: Unterstützung für ein neues Spreadsheet-Format

Neben der Unterstützung der bisherigen Excel-Dateiformate soll nun die Unterstützung für das OASIS Open Document Format for Office Applications (ODF) ermöglicht werden. Dafür wird eine externe Bibliothek, die verschiedene Lese- und Schreibfunktionen für ODF-Dateien bereitstellt, in das System integriert.

4.2.3 Ergebnisse

Da sich die Ergebnisse der beiden auf Szenarien basierenden Verfahren SAAM und ALMA zu großen Teilen überschneiden, werden hier die Ergebnisse der Evaluation der Szenarien zusammengefasst und vorgestellt. Zu den Unterschieden bei der Vorgehensweise und der Auswertung wird in Kapitel 9 näher eingegangen.

Szenario 1: Erstellen einer neuen Prüfung

Die Prüfung von Spreadsheets wird in den *Facilities* umgesetzt, das bedeutet für eine neue Prüfung wird auch eine neue *Facility* benötigt. Diese wird als neue Komponente erstellt und anschließend beim *TestBayManager* registriert. Dabei kann diese *TestFacility* die Klasse *MonolithicTestFacility* oder *CompositeTestFacility* erweitern, die den Zugriff auf die Daten und Metadaten über eine Schnittstelle bereitstellt.

Mit der Schnittstelle *IWorkbookCloner* können Kopien des bisherigen Spreadsheets angelegt werden. Die Änderung von Zellinhalten ist über die Schnittstelle *ISpreadsheetWriter* möglich. Mit der Schnittstelle *IDynamicSpreadsheetRunner* können dann Funktionen auf dem Spreadsheet ausgeführt werden.

Für die Konfiguration der benötigten Parameter wird eine neue *PolicyRule* angelegt. In *utilities.XML_Constants* werden dazu die benötigten Bezeichner für die Elemente der *PolicyRule* abgelegt. Anschließend wird die *PolicyRule* der *model.policy.PolicyList* hinzugefügt und bei den Datenaustauschobjekten *model.policy.Policy* und *model.policy.DynamicPolicy* registriert.

Zum Schluss wird eine neue Klasse erstellt, die das Interface *ISingleViolation* implementiert und als Datenaustauschobjekt für die Verstöße der Prüfungen verwendet wird.

Szenario 2: Unterstützung für ein neues Spreadsheet-Format

Um ein neues Spreadsheet-Format zu unterstützen, müssen die Schnittstellen *ISpreadsheetIO*, *IWorkbookCloner* und *ISpreadsheetWriter*, die den direkten Zugriff auf das Spreadsheet ermöglichen, für die neue Bibliothek implementiert werden. Außerdem müssen die Schnittstellen *IWorkbookCloner* und *ISpreadsheetWriter* angepasst werden, da sie bisher nur für die Verwendung der POI-Bibliothek geeignet sind. Zum Schluss müssen noch die Fallunterscheidungen im *InspectionManager* und *FrontDesk* implementiert werden, die abhängig vom Spreadsheet-Format, die korrekten Implementierungen der Schnittstellen aufrufen.

4.3 Metriken

Bei diesen Verfahren werden verschiedene Metriken der Architektur mit Hilfe von Werkzeugen erhoben. Grundlage für all diese Verfahren ist die statische Codeanalyse. Dabei werden der vorhandene Quellcode und/oder die Binärdateien analysiert und bestimmte Merkmale als Eingangsgrößen für die Metriken verwendet.

Als Erstes wird eine Auswahl an Metriken, die jeweils einzelne Merkmale der Architektur erfassen, mit der Entwicklungsumgebung IntelliJ IDEA erhoben. Dazu werden zuerst die verwendeten Metriken und die vermuteten Auswirkungen auf die Architektur definiert. Dann werden die Werte der Metriken durch Zählung oder Schätzung bestimmt und die daraus folgenden Eigenschaften der Architektur abgeleitet. Die Auswahl der Metriken stützt sich dabei auf die von Fowler et al. [FB99] ausgearbeiteten *Architektur-* und *Code-Smells*.

Als Nächstes wird mit den Werkzeugen Sonarqube und Teamscale die gesamte Architektur mit Hilfe von Metriken bewertet. Diese Werkzeuge stellen dafür eine Auswahl an Regeln bereit, die die Architektur und damit auch den Quellcode sowie Binärdateien erfüllen sollen. Auf Basis der Anzahl an Verstößen gegen diese Regeln, wird eine Bewertung der gesamten Software erstellt. Diese Metriken werden von Ludewig et al. [Lud13] als Pseudometriken bezeichnet, weil sie aus anderen geschätzten oder gezählten Metriken berechnet und nicht direkt gemessen werden können. Das Vorgehen zur Verwendung dieser Pseudometriken ähnelt jedoch der Verwendung anderer Metriken, ergänzt um einen zusätzlichen Schritt. Zuerst werden die benötigten Eingabegrößen, also die Werte der zu Grunde liegenden Metriken, bestimmt. Anschließend wird die Pseudometrik, nach dem vom Urheber vorgeschlagenen Modell, berechnet und interpretiert.

Zum Schluss wird eine *Dependency Structure Matrix Analysis* durchgeführt, bei der die Abhängigkeiten der einzelnen Komponenten in einer Matrix dargestellt werden.

4.3.1 Einzelmetriken

Die Anzahl verfügbarer Metriken für die Bewertung einzelner Merkmalen der Architektur ist sehr groß. Aus diesem Grund werden zuerst einige Metriken, die für eine solche Bewertung sinnvoll erscheinen, ausgewählt. Viele dieser Metriken sind subjektive Metriken, weil sie einen definierten

Schwellenwert verwenden, um eine Quantisierung des untersuchten Merkmals vornehmen zu können. Weil die Wahl des Schwellenwerts hauptsächlich von der vorhandenen Erfahrung des Entwicklers abhängt, lässt ein zu niedriger oder ein zu hoher Schwellenwert keine sinnvollen Aussagen mehr zu.

Definitionen der Metriken

Module mit vielen Komponenten

Java-Pakete mit mehr als 50 Klassen werden als Modul mit vielen Komponenten bezeichnet. Zu viele Komponenten innerhalb eines Moduls erhöhen die Komplexität des Moduls und können ein Anzeichen dafür sein, dass die Komponenten nicht gut genug strukturiert wurden.

Module mit wenigen Komponenten

Java-Pakete mit weniger als 5 Klassen werden als Modul mit wenigen Komponenten bezeichnet. Zu wenige Komponenten innerhalb eines Moduls führen zu einer großen Anzahl an Modulen, die wiederum die Komplexität der Architektur erhöhen. Viele Module mit wenigen Komponenten können ein Anzeichen dafür sein, dass die Architektur zu feingranular strukturiert wurde.

Maximale Verschachtelungstiefe von Modulen

Die maximale Verschachtelungstiefe von Modulen, wird als die Länge des längsten Wegs im Abhängkeitsgraphen der Module definiert. Wenn die Verschachtelungstiefe von Module hoch ist, kann das ein Anzeichen dafür sein, dass die Architektur zu feingranular strukturiert wurde.

Übermäßig gekoppelte Klassen

Wenn die Anzahl der referenzierten Klassen (ohne Systemklassen der *java.** Pakete) innerhalb einer Klasse den Schwellenwert von 15 übersteigt, wird diese als übermäßig gekoppelte Klasse bezeichnet. Übermäßig gekoppelte Klassen sind schwer wartbar und sollten aufgesplittet werden.

Klassen mit vielen Methoden

Wenn die Anzahl der Methoden einer Klasse den Schwellenwert von 20 übersteigt, wird diese als Klasse mit vielen Methoden bezeichnet. Eine hohe Anzahl an Methoden innerhalb einer Klasse deutet darauf hin, dass die Klasse zu viele verschiedene Aufgaben übernimmt und wahrscheinlich umstrukturiert werden sollte.

Verkettungen von *„instance of“* Prüfungen

Eine Verkettung von *„instance of“* Prüfungen ist die mehrmalige Nutzung des *„instance of“* Komparators innerhalb einer Verkettung von bedingten Anweisungen. Eine solche Verkettung weist oft darauf hin, dass die Prinzipien der Objektorientierung ignoriert wurden. Dies ist dann der Fall, wenn diese Verkettung dazu genutzt wird Typ-Prüfungen durchzuführen, anstatt eine entsprechende Spezialisierung zu verwenden. Typ-Prüfungen von externen Klassen werden nicht mitgezählt, da dort diese Verkettung oft die einzige Möglichkeit ist verschiedene Objekte einer nicht veränderbaren Elternklasse zu unterscheiden.

Markierungsschnittstellen

Markierungsschnittstellen (Marker Interfaces) sind Schnittstellen, die keine Methoden definieren und somit keine Funktionalität bereitstellen, sondern dazu genutzt werden Objekten Metainformationen hinzuzufügen. Diese Schnittstellen haben jedoch den Nachteil, dass die statische Typ-Prüfung des Compilers umgangen wird und das Vorhandensein dieser Schnittstellen nur zur Laufzeit mit dem *"instance of"* Komparators überprüft werden kann. Auch sind Markierungsschnittstellen für Menschen schwer zu verstehen, da die Funktionalität auf der Analyse von Metadaten beruht und somit der wahre Zweck verschleiert wird.

Singleton-Klassen

Singleton-Klassen sind Klassen, von denen immer nur eine einzige Instanz vorhanden ist und auf die global zugegriffen werden kann. Singleton-Klassen haben einige Nachteile und sollten deshalb nur mit Bedacht eingesetzt werden. So sollen Singleton-Klassen nur eine einzige Instanz besitzen, aber insbesondere bei Java und der Verwendung von Multithreading kann dies nicht immer technisch sauber umgesetzt werden und ist deshalb fehleranfällig. Auch steigt bei der übermäßigen Verwendung von Singleton-Klassen die Gefahr prozedural anstatt objektorientiert zu programmieren, wenn die Singleton-Klassen als Ersatz für globale Variablen verwendet werden.

Übermäßig gekoppelte Methoden

Wenn die Anzahl der referenzierten Klassen (ohne Systemklassen der *java.** Pakete) innerhalb einer Methode den Schwellenwert von 15 übersteigt, wird diese als übermäßig gekoppelte Klasse bezeichnet. Übermäßig gekoppelte Methoden sind schwer wartbar und sollten aufgesplittet werden.

Übermäßig verschachtelte Methoden

Wenn die Tiefe der Verschachtelung einer Methode den Schwellenwert von 4 übersteigt, wird diese als übermäßig verschachtelt bezeichnet. Übermäßig verschachtelte Methoden sind unübersichtlich, schwer zu verstehen und sollten aufgesplittet werden.

Methoden mit konstantem Rückgabewert

Methoden, bei denen der Rückgabewert eine Konstante ist, verschleiern, dass sie eigentlich nur Konstanten in Form von Methoden sind und sollten deshalb überprüft und überarbeitet werden. Oft sind diese Methoden Rückstände von Änderungen an der Architektur, die nicht korrekt implementiert wurden.

Verworfene Rückgabewerte von Methoden

Wenn die Rückgabewerte von Methoden, die häufig als Statusmeldungen der Ausführung implementiert werden, verworfen werden, dann ist das häufig ein Hinweis, dass die von der Architektur geforderten Kontrollflussmechanismen ignoriert werden.

Nicht verwendete Klassen

Klassen werden nicht verwendet, wenn diese implementiert aber im Programmablauf nie instanziert oder aufgerufen werden. Eine Architektur mit vielen, nicht verwendeten, aber vorhandenen Komponenten ist unübersichtlich und damit schwer wartbar.

Nicht verwendete Methoden

Methoden werden nicht verwendet, wenn diese implementiert aber im Programmablauf nie aufgerufen werden. Eine Architektur mit vielen, nicht verwendeten, aber vorhandenen Komponenten ist unübersichtlich und damit schwer wartbar.

Stille Exceptions

Als „stille Exception“ wird eine leere Anweisung innerhalb eines *catch*-Blocks beim Abfangen einer Exception bezeichnet. Stille Exceptions sollten unbedingt vermieden werden, weil sie Fehler verschleiern.

Anweisungen mit konstanten Bedingungen

Als Anweisung mit konstanter Bedingung wird eine Anweisung bezeichnet, bei der die Bedingung entweder stets oder nie erfüllt wird. Diese Anweisung erbringt keinen Nutzen, sondern verwirrt höchstens den Entwickler und sollte deshalb überarbeitet werden. Oft sind diese Anweisungen Rückstände von Änderungen an der Architektur, die nicht korrekt implementiert wurden oder das Resultat von falsch verstandener „defensiver Programmierung“ sind.

Ungenutzte Zuweisungen

Eine Zuweisung einer Variable wird dann nicht genutzt, wenn der Wert nach der Zuweisung nicht mehr gelesen wird. Eine solche Anweisung nimmt also keinen Einfluss auf den Programmablauf und sollte entfernt oder überarbeitet werden. Solche Zuweisungen können Hinweise auf eine fehlende oder fehlerhafte Flusskontrolle sein.

Tippfehler

Tippfehler sind alle Worte, die laut englischer oder deutscher Rechtschreibung einen Fehler enthalten. Eine hohe Anzahl an Tippfehlern kann auf eine mangelnde Qualitätskontrolle hinweisen. Auch können Tippfehler zu ungenutzten oder doppelt definierten Komponenten führen und sollten deshalb korrigiert werden.

Ergebnisse

Nach der Auswahl und der Definition werden die Werte der verwendeten Metriken bestimmt. Tabelle 4.1 listet die mit der Entwicklungsumgebung IntelliJ IDEA erhobenen Daten auf. Um den relativen Wert bestimmen zu können, wird noch die absolute Anzahl an Java-Packages (53), Klassen (208) und Funktionen (1122) bestimmt.

Metrik	Wert absolut	Wert relativ
Module mit vielen Komponenten	0	0%
Module mit wenigen Komponenten	36	67%
Maximale Verschachtelungstiefe von Modulen	7	-
Übermäßig gekoppelte Klassen	5	2,4%
Klassen mit vielen Methoden	4	1.9%
Verkettungen von *"instance of"* Prüfungen	19	-
Markierungsschnittstellen	4	-
Singleton-Klassen	3	1,4%
Übermäßig gekoppelte Methoden	4	0,3%
Übermäßig verschachtelte Methoden	5	0,3%
Methoden mit konstantem Rückgabewert	12	0,5%
Verworfene Rückgabewerte von Methoden	4	-
Nicht verwendete Klassen	22	10,5%
Nicht verwendete Methoden	257	22,9%
Stille Exceptions	2	-
Anweisungen mit konstanten Bedingungen	13	-
Ungenutzte Zuweisungen	36	-
Tippfehler	370	-

Tabelle 4.1: Werte der betrachteten Einzelmetriken vor dem Reengineering

4.3.2 Pseudometriken

Die Werkzeuge Teamscale und Sonarqube stellen für die Analyse der Softwarequalität eine Sammlung von Pseudometriken bereit, mit denen laut Aussage der Hersteller, einige Qualitätsmerkmale bestimmt werden können[1][2]. Die Auswahl der Werkzeuge erfolgte nach gründlicher Prüfung und vor allem aufgrund der Verfügbarkeit und des Funktionsumfangs der beiden Werkzeuge. Für beide Werkzeuge werden die standardmäßig vorgegebenen Einstellungen zur Bewertung von Softwareprojekten verwendet. Beide Werkzeuge untersuchen dabei den vorhandenen Java-Quellcode und die vom Compiler erzeugten Binärdateien. Da die Anforderungen der Werkzeuge an die Architektur nicht exakt mit jenen die an SIFCore gestellt werden, übereinstimmen, ist damit zu rechnen, dass einige *false-positives*, also fälschlicherweise erfasste Probleme auftreten, die die Bewertung verzerren. Die Werkzeuge verwenden Eingangswerte, welche sich auf Merkmale beziehen, die von einer Mehrheit der Java-Entwickler als *best practices* bezeichnet werden und zum Beispiel im CERT Oracle Secure Coding Standard for Java [Lon+11] spezifiziert sind. Es decken sich also auch viele Anforderungen die diese Werkzeuge an die Architektur stellen, mit denen die für SIFCore formuliert wurden und somit kann diese Analyse durchaus relevante Aussagen über die Architektur treffen.

[1] https://www.sonarqube.org/features/
[2] https://www.cqse.eu/de/produkte/teamscale/features/

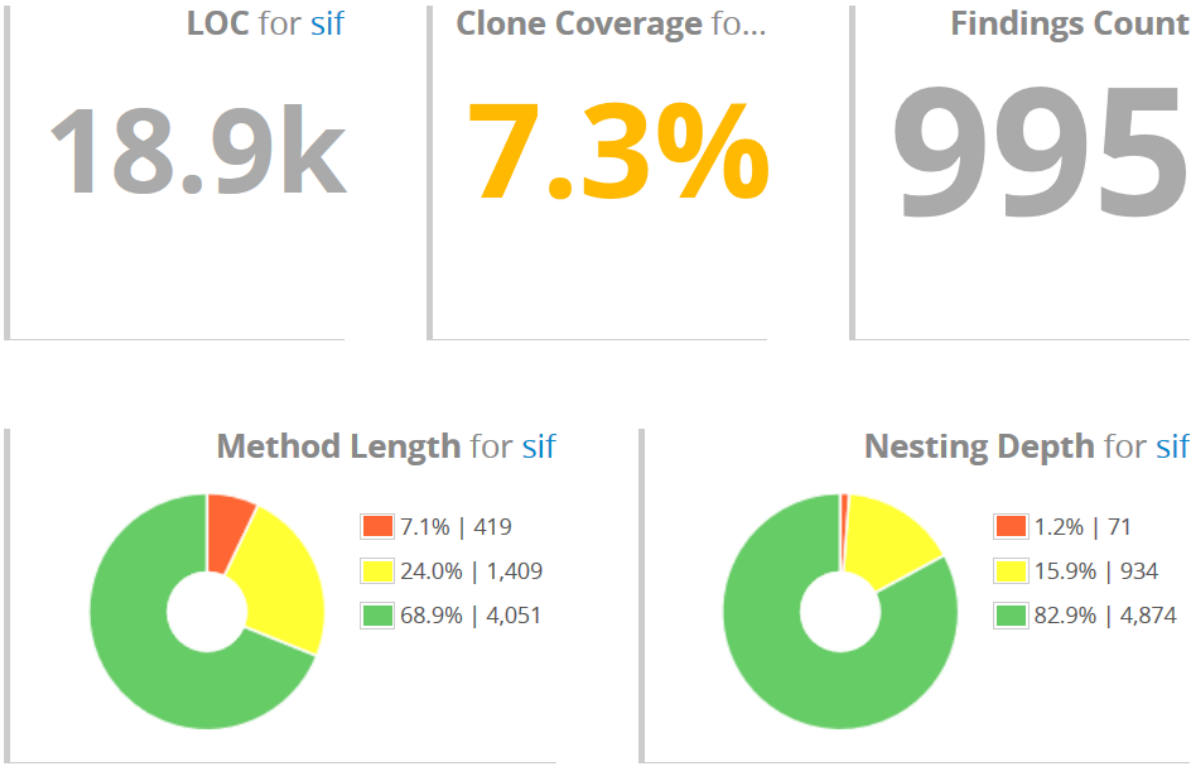

Abbildung 4.3: Das Dashboard von Teamscale für SIFCore vor dem Reengineering

Ergebnisse der Bewertung durch Teamscale

In Abbildung 4.3 wird das Dashboard der Teamscale-Benutzeroberfläche dargestellt. Teamscale nutzt grundsätzlich die drei Farben Grün, Gelb und Rot, um Werten von Metriken zu kennzeichnen. Grün bedeutet der Wert ist akzeptabel, Gelb, dass ein Problem besteht und Rot signalisiert ein schwerwiegendes Problem.

Die Metrik *Lines of Code (LoC)* zählt die Anzahl an Codezeilen des Projekts und soll somit eine Einschätzung über die Größe geben. Die Metrik *Clone Coverage* bestimmt den Prozentsatz an Zeilen im Quellcode, die über mindestens einen Klon (eine exakte Kopie) verfügen. Diese Metrik soll eine der häufigsten *„bad practices“*, die übermäßige Verwendung von *„copy&paste“* offenlegen. Die Metrik *Findings* ist die Anzahl der insgesamt von Teamscale entdeckten Probleme. Die Werte der beiden, als Ringdiagramme dargestellten Metriken, stellen die Länge (Method Length) und die Verschachtelungstiefe (Nesting Depth) aller in SIFCore vorhandenen Methoden dar. Ab einer Länge von 30 Zeilen oder ab einer Verschachtelungstiefe von 3 wird eine Methode als Problem eingestuft. Ab einer Länge von 75 Zeilen oder ab einer Verschachtelungstiefe von 5 wird eine Methode als schwerwiegendes Problem eingestuft.

In Abbildung 4.4 werden die einzelnen Probleme, aufgeschlüsselt nach Fehlergrad und Art des Fehlers, in einem Balkendiagramm dargestellt. Dabei sind Fehlerkategorien *Code Anomalies*, *Code Duplication* und *Structure* für die Bewertung der Architektur am interessantesten. Teamscale meldet für zwei der drei Kategorien mindestens einen schwerwiegenden Fehler. Die restlichen

Fehlerkategorien beziehen sich hauptsächlich auf die Qualität des Quellcodes und nicht auf die Architektur, sollten bei der Interpretation der Ergebnisse also entsprechend niedrig bewertet werden.

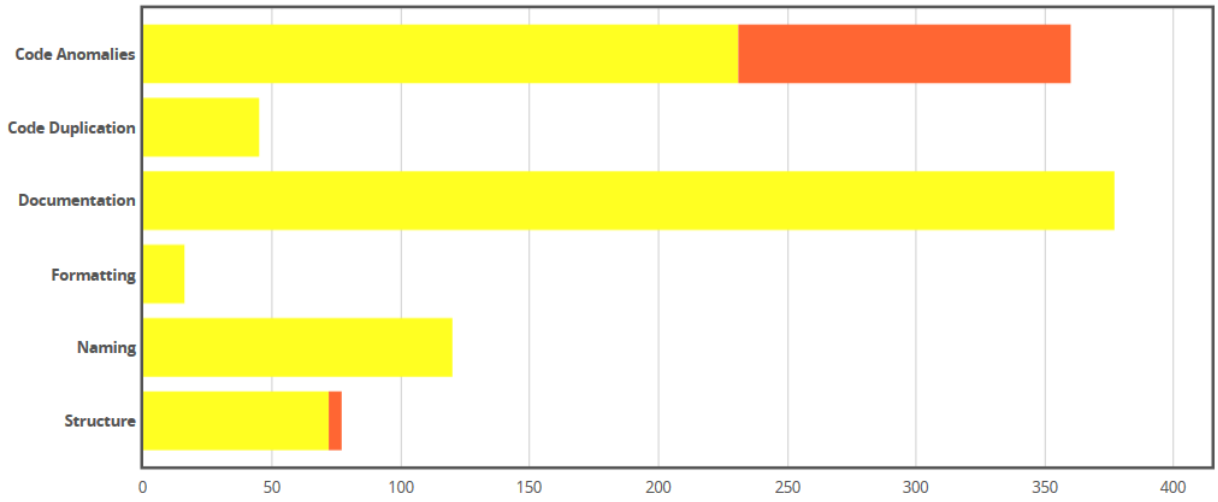

Abbildung 4.4: Von Teamscale gefundene Probleme von SIFCore vor dem Reengineering

Ergebnisse der Bewertung durch Sonarqube

In Abbildung 4.5 wird die Zusammenfassung der Bewertung von SIFCore durch Sonarqube dargestellt. Sonarqube bemisst dabei die drei Qualitätsaspekte Zuverlässigkeit, Wartbarkeit und Sicherheit. Die Bewertung erfolgt dabei mit einer Ordinalskala, die durch die Buchstaben A bis E gekennzeichnet wird. Ein „A“ steht für das bestmögliche und „E“ für das schlechteste Ergebnis. Neben dieser Bewertung der Qualitätsaspekte werden noch weitere Metriken angegeben, die eine eigene Interpretation der Ergebnisse oder eine eigene Bewertung des Softwareprojekts ermöglichen.

Unter dem Stichpunkt *Duplications* erfasst Sonarqube doppelt vorhandenen Quellcode. Dabei kann Sonarqube zwischen einzelnen kopierten Zeilen und ganzen zusammenhängenden Blöcken unterscheiden.

Unter dem Stichpunkt *Size* wird eine Sammlung von selbsterklärenden Metriken zur Bestimmung der Größe des Projekts dargestellt.

Unter dem Stichpunkt *Complexity* wird die Pseudometrik Komplexität für das Gesamtprojekt und einzelne Dateien, Klassen oder Funktionen angegeben. Bei der Berechnung der Komplexität werden die Vorkommen der Schlüsselwörter *if*, *for*, *while*, *case*, *catch*, *throw*, *return*, *&&*, *||* und *?* gezählt und für jedes Vorkommen wird die Komplexität inkrementiert.

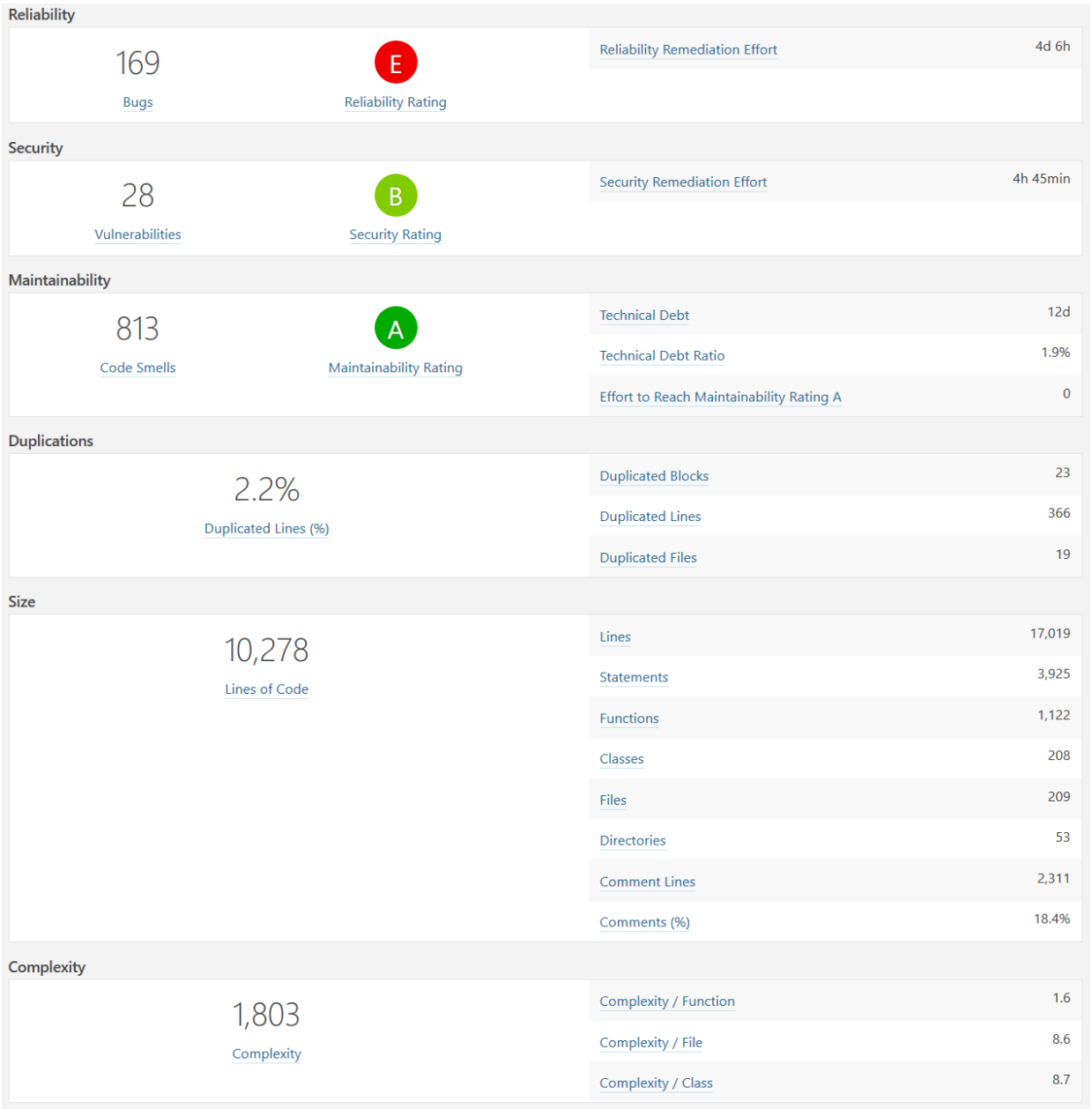

Abbildung 4.5: Bewertung von SIFCore durch Sonarqube vor dem Reengineering

4.3.3 Dependency Structure Matrix Analysis

Bei der *Dependency Structure Matrix Analysis* werden die einzelnen Abhängigkeiten der Komponenten einer Architektur in einer Matrix dargestellt. Diese, ursprünglich aus Geschäftsprozessmanagement stammende Technik [Ste81] wurde von Sangal et al. [San+] auf die Softwareentwicklung übertragen.

Bei der DSM repräsentieren die Komponenten jeweils die Zeilen und Spalten der Matrix. Es wird also eine Matrix A der folgenden Form aufgestellt:

$$A : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathbb{N}_0, \quad (i, j) \mapsto a_{ij}$$

Dabei ist *n* die Anzahl an Komponenten und die Funktion $(i, j) \mapsto a_{ij}$ liefert als Funktionswert die Anzahl der Abhängigkeiten der Komponente j zur Komponente i. Es werden also die Abhängigkeiten der Spalten auf die Zeilen eingetragen und nicht anders herum. Wenn keine Abhängigkeit besteht wird kein Wert eingetragen. Die Funktion gibt auch für $i = j$ keinen Wert zurück, folglich befinden sich auf der Hauptdiagonale der Matrix keine Einträge.

Komponenten sind im Falle von SIFCore die Java-Pakete und Klassen Eine Abhängigkeit zu anderen Komponenten wird über die *import* Anweisung innerhalb einer Klasse im Quelltext definiert. Nachdem alle Abhängigkeiten der Komponenten gezählt und in die Matrix eingetragen wurden, wird die Matrix anschließend sortiert. Für die Sortierung wird für jede Zeile und Spalte die Summe der eingetragenen Abhängigkeiten bestimmt. Dann werden die Komponenten nach dem Kriterium *niedrigste Summe einer Spalte* und bei gleichem Wert nach dem Kriterium *niedrigste Summe einer Zeile* sortiert. Dadurch wandern die meisten Einträge unter die Hauptdiagonale und es werden (sofern vorhanden) die Schichten der Architektur sichtbar.

Weit oben (von den Zeilen der Matrix ausgehend) stehen die Komponenten, die nur Abhängigkeiten zu anderen Komponenten haben und dementsprechend auch weit oben in einer Schichtenarchitektur angesiedelt sind. Unten finden sich hauptsächlich Komponenten, die als Abhängigkeiten für viele andere Komponenten benötigt werden, aber selbst kaum eigene Abhängigkeiten haben. Für eine ideale Architektur ohne zyklische Abhängigkeiten nimmt die DSM die Form einer unteren Dreiecksmatrix an. Diese Matrix kann dann auch als gerichteter azyklischer Graph dargestellt werden.

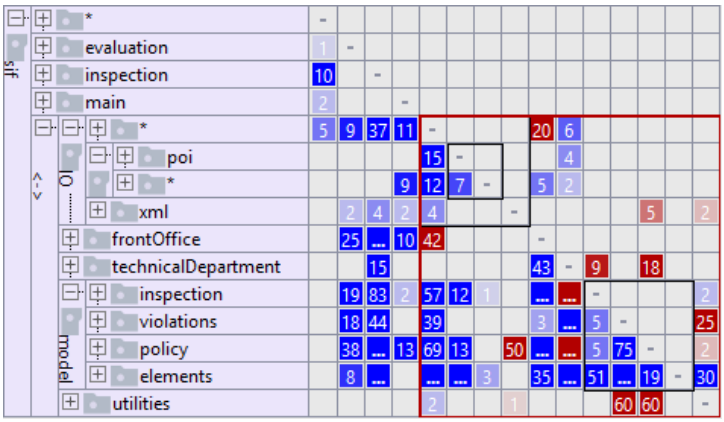

Abbildung 4.6: Dependency Structure Matrix von SIFCore vor dem Reengineering

Mit Hilfe von IntelliJ IDEA kann die in Abbildung 4.6 gezeigte DSM von SIFCore erzeugt werden. Dabei ist zu beachten, dass die Matrix nicht vollständig dargestellt wird, sondern einige Komponenten (so zum Beispiel alle Klassen) ihren übergeordneten Komponenten zugerechnet werden. Auf eine Darstellung der vollständigen und damit sehr großen DSM wird hier verzichtet, diese befindet sich auf der beigelegten CD im SIFCore-Projektarchiv.

In der Darstellung der DSM werden die Elemente der Matrix unterschiedlich eingefärbt. Je größer der Wert der Abhängigkeiten für ein Element ist, desto dunkler wird es eingefärbt. Zudem werden direkte, zyklische Abhängigkeiten, das heißt wenn Komponente A von Komponente B abhängt oder umgekehrt, rot markiert. Es werden alle Komponenten, bei denen bereits eine untere Dreiecksmatrix vorliegt mit einem schwarzen Rahmen, und alle Komponenten zwischen denen zyklische Abhängigkeiten bestehen, mit einem rotem Rahmen markiert.

4.4 Simulationen und Experimente

Für die Auswertung des öffentlichen Issue-Trackers werden alle Anliegen der Benutzer, die sogenannten Tickets, erfasst und kategorisiert. Diese Tickets können Fehlermeldungen enthalten, können aber auch für Feature-Wünsche, Kommentare oder Support-Anfragen genutzt werden. Ganz allgemein dienen Issue-Tracker dazu, Anliegen der Entwickler und Benutzer der Software zu kategorisieren und zu archivieren. Da das SIF inzwischen aus den zwei Unterprojekten SIFCore und SIFEI besteht und der Issue-Tracker gemeinsam für beide Projekte genutzt wird, muss zuerst festgestellt werden, ob das erstellte Ticket überhaupt SIFCore betrifft. Des Weiteren muss aufgrund der hohen Anzahl von 55 Tickets eine weitere Unterteilung erfolgen. Dafür wird eine Taxonomie erstellt, die die Tickets in die folgenden Kategorien einordnet:

- **Schwerwiegender Fehler**: Ein Fehler, der eine Ausführung des Programms unmöglich macht.
- **Fehler** Ein Fehler, der den Programmablauf beeinflusst, aber nur unter bestimmten Bedingungen auftritt oder den Programmablauf nicht unterbricht.
- **Geringfügiger Fehler**: Es geringfügiges Probleme, wie zum Beispiel eine falsche oder nicht vorhandene Übersetzung.
- **Feature**: Ein Anliegen, das eine Änderung der funktionalen oder nicht-funktionalen Anforderungen von SIFCore benötigt.
- **Support**: Ein Anliegen, das Fragen zur Bedienung oder Funktionalität beinhaltet.
- **Unklar**: Ein Anliegen, das zu wenig Information bietet, um es zu kategorisieren oder bearbeiten zu können.
- **SIFEI**: Ein Anliegen, das ausschließlich SIFEI betrifft. Diese wird nicht weiter kategorisiert und bei der weiteren Analyse ignoriert.

Nach der Kategorisierung werden die für eine Architekturanalyse relevant erscheinenden Tickets näher untersucht. Das sind vor allem die Tickets der Kategorien *schwerwiegender Fehler* und *Feature*. Dabei werden zuerst Tickets geprüft, die explizit die Architektur betreffen und dann die Tickets die nur einen indirekten Bezug zur Architektur haben. Es wird für jedes Ticket abgeklärt in welchem Umfang für die Behebung eines Fehlers oder die Umsetzung eines Features die Architektur geändert werden muss oder aber welche Komponenten der Architektur von diesem Anliegen betroffen sind. Ausgehend davon wird am Ende durch eine Interpretation der Ergebnisse auf den aktuellen Zustand der Architektur geschlossen.

S. Fehler	Fehler	G. Fehler	Feature	Support	Unklar	SIFEI	**Gesamt**
0	2	0	7	0	3	43	55

Tabelle 4.2: Anzahl der Tickets in den jeweiligen Kategorien

In Tabelle 4.2 werden die Ergebnisse der Kategorisierung dargestellt. Nach der Kategorisierung werden fünf Tickets der Kategorien *Fehler* und *Feature* als relevant für die weitere Analyse eingestuft. Die Ergebnisse dieser Analyse werden in Tabelle 4.3 zusammengefasst.

4.5 Durchsicht

Als letztes Verfahren wird eine Durchsicht der in Kapitel 3 erstellten Architekturdokumentation angewendet. Dabei handelt es sich um eine manuelle Inspektion eines Prüflings von einer einzelnen Person. Laut Ludewig et al. [Lud13] wird dieses Verfahren am besten abseits des Rechners durchgeführt, damit der Prüfling aus einer gewissen Distanz betrachtet werden kann. Es werden bei diesem Verfahren keine weiteren formalen Regeln aufgestellt und auch die Form, in der die Ergebnisse dieser Prüfung präsentiert werden, ist nicht festgelegt.

Bei der Durchsicht der Architekturdokumentation von SIFCore soll jedoch nicht die Qualität der Dokumente überprüft werden, sondern die Qualität der Architektur, die damit dokumentiert wurde. Aufgrund des informellen Charakters einer Durchsicht, ist es schwierig die Ergebnisse in sinnvoller Weise zu präsentieren, weil sie vorwiegend aus handschriftlichen Notizen bestehen. Aus diesem Grund wird auf eine gesonderte Darstellung der Ergebnisse verzichtet. Stattdessen werden sie direkt in die Bewertung der Architektur im nächsten Abschnitt einfließen und dort dann entsprechend gekennzeichnet.

4.6 Bewertung der Architektur

Die bisher erhobenen Ergebnisse werden nun interpretiert, damit eine Bewertung der Architektur vorgenommen werden kann. Diese ist stark vom jeweiligen Entwickler geprägt, weil die Interpretation der Ergebnisse von den persönlichen Vorlieben, Erfahrungswerten und dem Wissen des Entwicklers abhängig ist. In dieser Bewertung werden einige Schwachstellen der bisherigen Architektur herausgearbeitet, die dann im nächsten Kapitel verbessert werden.

ID	Kategorie	Beschreibung	Implikationen
134	Fehler	Wenn bei der Ausführung einer Prüfung in einem Szenario eine Nummer im Exponentialdarstellung verwendet wird bricht die Ausführung des Programms mit einer Exception des XMLParsers ab.	Die *PolicyIO* Komponente scheint nicht robust gegen die Eingabe von ungültigen Werten zu sein. Zusätzlich wird der Fehler von der *TestBayManager* Komponente offensichtlich nicht korrekt abgefangen und verarbeitet, da er zu einem Abbruch des Programmablaufs führt.
96, 53	Feature	Im Moment wird SIFCore über ein Exec-Befehl von SIFEI gestartet und dann Verbindet sich SIFCore über einen Socket als Client wieder mit SIFEI. Dies erschwert jedoch das Debugging im Testbetrieb, es wäre also wünschenswert wenn SIFEI eine Socketverbindung zu einer laufenden SIFCore-Instanz aufbauen könnte.	Dieser Feature-Wunsch erfordert große Änderungen an der Architektur von SIFCore und SIFEI, denn es müssen die Rollen von Client und Server bei der Socket-Kommunikation vertauscht werden. Im Moment könnte zwar SIFCore unabhängig von SIFEI gestartet werden, dann ist aber keine Kommunikation möglich, da die Verbindung der Socket-Kommunikation von Client gestartet werden muss.
43	Fehler	Es existieren für einige Fehlermeldungen Übersetzungen, aber für andere aber nicht.	Die Architektur unterstützt grundsätzlich Internationalisierbarkeit, diese wird jedoch nicht konsequent umgesetzt. Diese Fehlerbehebung erfordert wahrscheinlich keinen großen Änderungen an der Architektur, da die Unterstützung bereits vorhanden ist. Es kann jedoch Einzelfälle geben in denen die bisherigen Möglichkeiten zur Übersetzung oder Formatierung nicht ausreichen, sodass in diesem Fall durchaus Änderungen an der Architektur vorgenommen werden müssen.
37	Fehler	Es wird bemängelt, dass die Prüfung von SIFCore sehr unzulässig ist und zufällig abstürzt. Leider gibt es dazu keine weiteren Metriken oder Hinweise.	Da keine weiteren Informationen verfügbar sind, ist nur schwer auszumachen wie unzuverlässig die Prüfung von SIFCore wirklich ist. Aber scheinbar zufällige Fehler und Abstürze lassen auf ein Architekturproblem schließen.

Tabelle 4.3: Ergebnisse der Analyse des Issue-Trackers

Die Auswertung des Issue-Trackers wurde als erstes Verfahren durchgeführt, da die dafür erforderlichen Informationen von Anfang an zur Verfügung standen und die Durchführung dieser Aufgabe vorgeschrieben war. Die Ergebnisse der Analyse weisen gleich auf ein der zentrales Problem der Architektur hin: Viele Erweiterungen von SIFCore wurden nicht korrekt in die Architektur eingefügt und implementiert. Für diese Annahme spricht, dass alle gemeldeten Probleme nur Komponenten betreffen, die in der ersten Version von SIFCore nicht vorhanden waren. Dazu gehören die Socket-Kommunikation, der XML-Parser und die Übersetzungsfunktion.

Die Frage, warum gerade diese Komponenten Probleme hervorrufen, ist nur schwer zu beantworten, da viele Faktoren eine Rolle spielen können. Aber die Möglichkeit, dass eine unflexible Architektur diese Probleme mit verursacht, muss bedacht werden. Ein Grund hierfür könnte die zentrale Metapher von SIFCore sein. Bei der Ursachenforschung im Rahmen der Durchsicht fällt auf, dass die späteren Iterationen, die in der ersten Arbeit [Zit12] propagierte Metapher von SIFCore zwar nennen, sie aber im Entwurf und der Implementierung weitestgehend ignorieren. Die Metapher wirkt zu speziell und zu detailliert, um für die Entwickler hilfreich zu sein.

Ein weiteres Problem der Architektur ist die wohl nur unzureichend geplante Kommunikationskomponente. So sind kaum Informationen oder Beschreibungen zu dieser Komponente zu finden. Es ist lediglich eine kurze Begründung vorhanden [Dou13], dass eine Socket-Kommunikation auf dem Betriebssystem Windows unterstützt wird und diese leicht zu implementieren ist. Das bei der Analyse des Issue-Trackers gefundene Problem durch vertauschte Server- und Client-Rollen bestärkt dabei die Annahme der unzureichenden Planung dieser Komponente. Die Änderungen der Architektur zur Behebung dieses Problems wären sehr umfangreich, daher ist es zu überlegen, ob die Komponente nicht besser komplett neu entworfen und implementiert werden sollte.

Bei der Auswertung der Szenarien für die Analyseverfahren ALMA und SAAM fallen ebenfalls sofort einige Probleme der Architektur auf. So unterstützt die ursprüngliche Architektur zwar das Erstellen neuer Prüfungen, welches eine der nicht-funktionalen Anforderungen war. Jedoch erfüllen die später hinzugefügten Komponenten diese nicht. Die Komponenten (Facility, Policy und Violation), die für eine Prüfung benötigt werden, sind auf unterschiedliche Module verteilt, obwohl doch eine enge Kopplung zwischen diesen Komponenten besteht. Auch ist es unklar, warum es die in der ersten Iteration eingeführten *CombinedTestFacilites* noch gibt, denn diese wurden mit der Einführung des SpRuDeL-Austauschformats nicht mehr benötigt.

Darüber hinaus ist die Verwendung von zwei verschiedenen Verwaltungskomponenten für die Prüfungskonfiguration (*Policy* und *DynamicPolicy*) umständlich, gar unnötig. Es scheint mir, als wäre der Aufwand für ein ordentliches Überarbeiten der *Policy*-Komponente gescheut und stattdessen die Komponente mittels Spezialisierung erweitert worden. In einigen Komponenten wird noch die alte Basisklasse verwendet, in anderen dagegen, die neue Spezialisierung. Dies ist eindeutig eine Verletzung des SRP und hat zur Folge, dass viele Daten doppelt gespeichert oder bei zwei verschiedenen Komponenten registriert werden müssen.

Bei der Analyse des zweiten Szenarios fällt auf, dass die in späteren Iterationen erstellten Schnittstellen zur dynamischen Bearbeitung und Ausführung von Spreadsheets, ungünstig umgesetzt sind. So stellt sich die Frage, warum überhaupt drei neue Schnittstellen (*IWorkbookCloner*, *ISpreadsheetWriter* und *IDynamicSpreadsheetRunner*) eingeführt wurden, anstatt die bereits bestehende Schnittstelle *SpreadsheetIO* zu erweitern, die bereits für die Kommunikation zwischen Modell und

Spreadsheet-Bibliothek vorgesehen war. Zusätzlich sind diese Schnittstellen von der eingesetzten Bibliothek abhängig und sind somit nicht als unabhängige Schnittstellen für andere Spreadsheet-Formate geeignet. Wie die Durchsicht der Architekturdokumentation gezeigt hat, sind diese Schnittstellen und ihre jeweiligen Implementierungen in verschiedene Module strukturiert und generell uneinheitlich umgesetzt. Die Änderungen der späteren Iterationen wirken wie Fremdkörper in der Architektur und es liegt die Vermutung nahe, dass viele Komponenten einfach neu erstellt wurden, um ältere (und funktionsfähige) Komponenten nicht ändern zu müssen.

Bei der Interpretation der Ergebnisse der DSM-Analyse fällt sofort die, meiner Meinung nach, recht hohe Zahl an zyklischen Abhängigkeiten auf. Zyklische Abhängigkeiten müssen nicht immer auf Probleme der Architektur hinweisen und teilweise sind diese auch nur der jeweiligen Implementierung geschuldet. Bei zyklischen Abhängigkeiten in dieser Anzahl, die zwischen so vielen verschiedenen Modulen besteht, muss ich zum Schluss kommen, dass einige Fehler bei der Strukturierung der Software gemacht wurden. Gleichzeitig zeigt die DSM, dass auch sehr viele normale Abhängigkeiten unter den Modulen bestehen. Dies bedeutet, dass die Kopplung zwischen den Modulen hoch ist und lässt den Schluss zu, dass der Zusammenhalt der Module relativ gering ist. Die Aufgaben der einzelnen Module sind also nicht klar gegeneinander abgegrenzt und möglicherweise in zu viele einzelne Module aufgesplittet. Damit ist es auch wahrscheinlich, dass die gewählten Strukturen der Komponenten zu fein und damit kontraproduktiv sind. Die Durchsicht der Architekturdokumentation bestätigt meine Vermutung an vielen Stellen. Generell ist die Anzahl der Java-Pakete sehr hoch, so zeigen die durch Sonarqube bereitgestellten Metriken, dass 208 Java-Klassen auf 53 Java-Pakete verteilt werden.

Bei der Betrachtung, der von mir ausgewählten einzelnen Metriken, fallen besonders die Metriken „Module mit wenigen Komponenten“, „Nicht verwendete Klassen“ und „Nicht verwendete Methoden“ mit hohen relativen Werten von 67%, 10,5% und 22,9% auf. Dies unterstützt meine These, dass die Architektur von SIFCore „über“-strukturiert ist. Ebenso wurden alte, nicht mehr benötigte Komponenten nicht entfernt und somit wurde die Struktur weiter verkompliziert.

Als Konsequenz aus dieser Bewertung und den eben genannten Problemen der Architektur, werden die folgenden Maßnahmen als Teil des Reengineerings bestimmt:

- Eine Restrukturierung aller Module, um einen starken Zusammenhalt und eine geringe Kopplung zwischen den Modulen zu gewährleisten.
- Eine gründliche Überarbeitung der Kommunikations-Komponente und des damit verbundenen Austauschformats.
- Die Überarbeitung oder Entfernung der zentralen Metapher, weil sie derzeit nicht nützlich ist.
- Die Entfernung oder Verbesserung der Komponenten, die die Schichten der Architektur verletzen und somit problematisch für die Einführung eines neuen Spreadsheet-Formats sind.
- Die Überarbeitung der Schnittstellen, die für die dynamische Änderung und Ausführung der Spreadsheets verantwortlich sind.
- Die Überarbeitung der Fehlerbehandlung und die Verbesserung der Robustheit von SIFCore.

5 Anforderungen

Nachdem im vorherigen Kapitel die bisherige Architektur des SIF analysiert und bewertet wurde, werden die Anforderungen an die neue Architektur formuliert. Die Anforderungen an SIFCore haben sich zwar nur in geringem Umfang gegenüber den Vorgängerversionen geändert, eine vollständige und aktuelle Auflistung fehlt aber. In diesem Kapitel werden deshalb die bisherigen Anforderungen zusammengefasst, aktualisiert und bei Bedarf überarbeitet.

5.1 Funktionale Anforderungen

Die funktionalen Anforderungen an SIFCore ändern sich zu großen Teilen nicht. Lediglich die Kommunikation von SIFCore zu SIFEI wird überarbeitet. Grundlage dieser funktionalen Anforderungen sind die von Sebastian Zitzelsberger formulierten Anforderungen ([Zit12]). Die Anforderungen für die einzelnen Prüfungen werden aus den Beschreibungen in den jeweiligen Arbeiten ([Lem13], [Dou13], [Kra14]) abgeleitet, soweit diese dort vorhanden sind. Ansonsten werden die Anforderungen auf Basis der verfügbaren Informationen und der Implementierung neu formuliert.

5.1.1 Einlesen von Spreadsheets in Microsoft Office Formaten

In SIFCore soll der Import aller von Microsoft verwendeten Spreadsheet-Formate unterstützt werden. Die Unterstützung soll mit Hilfe der Apache POI-Bibliothek ermöglicht werden, mit der Spreadsheets in ein geeignetes Modell aus Java-Datenobjekten überführt werden können.

5.1.2 Erstellen von zusätzlichen Metadaten durch Analyse des Modells

Nach dem Import eines Spreadsheets und der Erstellung des Modells soll es möglich sein das Modell mit zusätzlichen Metainformationen bestimmter Objekte zu versehen.

5.1.3 Durchführung von statischen und dynamischen Prüfungen

Mittels der Daten und Metadaten, die im internen Modell gespeichert sind, sollen Prüfungen bestimmter Regeln, sowie der Einhaltung von Eigenschaften des Spreadsheets, ermöglicht werden. Dabei sollen sowohl statische Eigenschaften von Spreadsheets geprüft, als auch die Veränderung und Ausführung des Spreadsheets ermöglicht werden.

5.1.4 Verwendung eines auf XML basierenden Austauschformats

Die Konfiguration der Prüfungen, sowie die Auflistung aller gemeldeten Verstöße der Prüfungen, sollen mit Hilfe eines auf XML basierenden Austauschformats umgesetzt werden.

5.1.5 Prüfung: Szenarien

Mit Hilfe von Szenarien sollen Prüfungen für Spreadsheets ermöglicht werden, die vergleichbar sind mit den Modultests von Software. Es sollen für die Szenarien vom Benutzer Eingangswerte, Invarianten und Ausgangswerte für die Formeln des Spreadsheets definiert werden. Diese Bedingungen werden dann bei der Ausführung des Spreadsheets überprüft.

5.1.6 Prüfung: Formeln mit Fehlern

Bei dieser Prüfung sollen alle Fehler, die in Formeln des Spreadsheets enthalten sind, erkannt werden. Alle von Microsoft Excel erkannten Fehler werden auch in SIFCore als Fehler definiert.

5.1.7 Prüfung: Formelkomplexität

Bei dieser Prüfung sollen alle Formeln auf ihre Verschachtelung und Komplexität geprüft werden. Die Definition dieser Begriffe wird über die Implementierung dieser Prüfung implizit festgelegt. Die Grenzwerte, ab denen die Prüfung einen Verstoß meldet, sollen vom Benutzer festgelegt werden können.

5.1.8 Prüfung: Gleiche Verweise

Bei dieser Prüfung sollen alle Formeln und die darin enthaltenen Verweise auf andere Zellen untersucht werden. Mehrere gleiche Referenzen hintereinander sollen als Verstoß gemeldet werden.

5.1.9 Prüfung: Konstanten in Formeln

Bei dieser Prüfung sollen alle Formeln überprüft werden. Jede Konstante in einer Formel, also die Verwendung eines statischen Werts anstatt einer Referenz auf eine Zelle, soll dabei als Verstoß gemeldet werden.

5.1.10 Prüfung: Nicht beachtete Werte

Bei dieser Prüfung sollen alle Zellen des Spreadsheets auf ihre Werte hin überprüft werden. Wenn diese Werte in keiner Formel referenziert werden, soll ein Verstoß gemeldet werden. Der Benutzer soll konfigurieren können, ob er nur numerische Werte oder auch Zeichenketten untersuchen möchte.

5.1.11 Prüfung: Zellennachbarschaft

Bei dieser Prüfung soll die Umgebung von Werten auf dem Spreadsheet untersucht werden. Der Benutzer kann die Parameter der Umgebung dabei selbst bestimmen. Wenn in der Umgebung einer Zelle, keine Zellen mit dem gleichen Typ an Werten vorhanden ist, soll ein Verstoß gemeldet werden.

5.1.12 Prüfung: Leserichtung

Bei dieser Prüfung sollen die Referenzen von Formeln im Spreadsheet untersucht werden. Wenn die Referenzen nicht einer vom Benutzer konfigurierbaren Regel bezüglich ihrer Platzierung auf dem Spreadsheet folgen, soll ein Verstoß gemeldet werden.

5.1.13 Prüfung: Referenzen auf leere Zellen

Bei dieser Prüfung sollen die Referenzen von Formeln im Spreadsheet untersucht werden. Wenn die Referenzen auf leere Zellen zeigen, soll ein Verstoß gemeldet werden.

5.1.14 Prüfung: Levenshtein-Distanz

Bei dieser Prüfung soll die Levenshtein-Distanz zwischen Zellen, die Zeichenketten beinhalten, untersucht werden. Wenn die Distanz einen vom Benutzer konfigurierbaren Wert unterschreitet, soll ein Verstoß gemeldet werden.

5.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen wurden bisher weder eindeutig, noch vollständig formuliert. Lediglich Sebastian Zitzelsberger [Zit12] nennt explizit drei nichtfunktionale Anforderungen. Die hier erarbeiteten nichtfunktionalen Anforderungen werden aus dem Kontext der bisherigen Arbeiten, insbesondere den Aufgabenstellungen und den Ergebnissen der Architekturrekonstruktion und der Architekturanalyse abgeleitet. Zur Kontrolle auf Vollständigkeit, wird der im Kapitel 2 beschriebene *Qualitätenbaum* verwendet und um die dort nicht aufgeführten Qualitätsmerkmale *Security* und *Safety* ergänzt.

5.2.1 Prüfbarkeit

Es soll sichergestellt werden, dass Anforderungen an SIFCore mit Modul- und Integrationstests überprüft werden können. Die Architektur soll dabei so gestaltet sein, dass sich einzelne Module oder Komponenten leicht testen lassen. Dabei sollte eine Testabdeckung der Komponenten von mindestens 80% erreicht werden.

5.2.2 Änderbarkeit

SIFCore soll sich leicht um weitere Prüfungen von Spreadsheets erweitern lassen. Das bedeutet, dass der Einarbeitungs- und Zeitaufwand dafür möglichst gering sein sollte und die Struktur der Architektur eine solche Änderung unterstützt.

5.2.3 Portabilität

Bei der Implementierung soll darauf geachtet werden, dass SIFCore mit möglichst vielen Plattformen und Betriebssystemen verwendet werden kann. Voraussetzung dafür soll lediglich eine vorhandene Java Virtual Machine (JVM) für das aktuelle Java 8 sein. Gefordert wird eine Unterstützung für die Plattform x86-64 (AMD64) mit den Betriebssystemen Windows 7,8 und 10 sowie Linux mit Kernel 3.x und 4.x.

5.2.4 Zuverlässigkeit

SIFCore soll die Prüfung von Spreadsheets zuverlässig und schnell durchführen. Dabei soll sichergestellt werden, dass mindesten 1000 unabhängige Prüfungen nacheinander und 10 Prüfungen gleichzeitig durchgeführt werden können, ohne dass Einschränkungen bei der Zuverlässigkeit hingenommen werden müssen. Es soll sichergestellt sein, dass SIFCore nicht durch fehlerhafte oder unvollständige Anfragen zum Absturz gebracht werden kann.

5.2.5 Nützlichkeit

SIFCore soll alle geforderten funktionalen Anforderungen erfüllen und konsistente und nachvollziehbare Ergebnisse liefern. Die Erfüllung dieser Anforderung soll durch entsprechende Integrationstests der Prüfungen nachgewiesen werden.

5.2.6 Bedienbarkeit

Da SIFCore nur im Rahmen von Machine-to-Machine (M2M) benutzt wird, werden keine besonderen Anforderungen an die Bedienbarkeit gestellt.

5.2.7 Safety

Da SIFCore nicht in einer Umgebung genutzt werden soll, in der es besondere Safety-Anforderungen gibt, werden in dieser Hinsicht keine Forderungen gestellt.

5.2.8 Security

Die Kommunikation von SIFCore soll so gestaltet sein, dass sie in späteren Versionen um eine Authentifizierungsebene und eine Zugriffsebene erweitert werden kann. Ansonsten werden keine besonderen Anforderungen an die Sicherheit von SIFCore gestellt.

6 Architekturentwurf

In diesem Kapitel werden nun, die im vorherigen Kapitel formulierten Anforderungen in einem Entwurfsprozess als neue Architektur von SIFCore umgesetzt und dabei so beschrieben, dass anschließend direkt mit der Implementierung fortgefahren werden kann.

6.1 Konzepte

Um einige der Anforderungen einfacher erfüllen zu können, werden einige neue Konzepte für die Architektur von SIFCore eingeführt. Dabei handelt es sich um den Architekturstil *serviceorientierte Architektur* (SOA), das Programmierparadigma *Representational State Transfer* (REST) und das Entwurfsmuster *Dependency Injection* (DI).

6.1.1 Serviceorientierte Architektur

Durch die Verwendung einer SOA soll SIFCore modernisiert, flexibilisiert und vereinfacht werden. Um die SOA umzusetzen, wird die Prüfung von Spreadsheets ab sofort als *Service* oder *Dienst* von SIFCore betrachtet. SIFEI ist dabei einer der möglichen Nutzer dieses Dienstes. Die Prüfung eines Spreadsheets erfolgt über eine Anfrage des Clients an den entsprechenden Endpunkt, der auch Ressource genannt wird. Eine Übersicht über die verwendeten Komponenten zur Umsetzung der SOA wird in Abbildung 6.1 dargestellt.

Um den Dienst möglichst abgeschlossen zu gestalten, wird dieser in Form eines Java-Servlet umgesetzt. Dieses Servlet kann dann mit einem geeigneten Web- oder Applicationserver ausgeführt werden. Damit sich der Aufwand für ein Deployment in Grenzen hält, wird in SIFCore ein eigener Webserver integriert, der das Servlet direkt ausführen und veröffentlichen kann. Durch diese Strukturierung kann bei Bedarf das Servlet mit einem anderen Applikationsserver ausgeführt oder in einer Cloud-Umgebung eingesetzt und so mit weiteren Diensten vernetzt werden.

6.1.2 Representational State Transfer

Es wird bei der Umsetzung von SOA auf das, bereits in Kapitel 2 beschriebene, Programmierparadigma REST gesetzt. Die Anfragen des Clients, im Falle von SIF der Visualisierungsclient SIFEI, werden mittels HTTP an den Dienst, in diesem Fall SIFCore, übertragen. Dabei enthält die Anfrage (engl. Request) alle für die Prüfung benötigten Informationen. Die Nutzdaten der Anfrage enthalten die Konfiguration der Prüfungen, sowie das Spreadsheet das geprüft werden

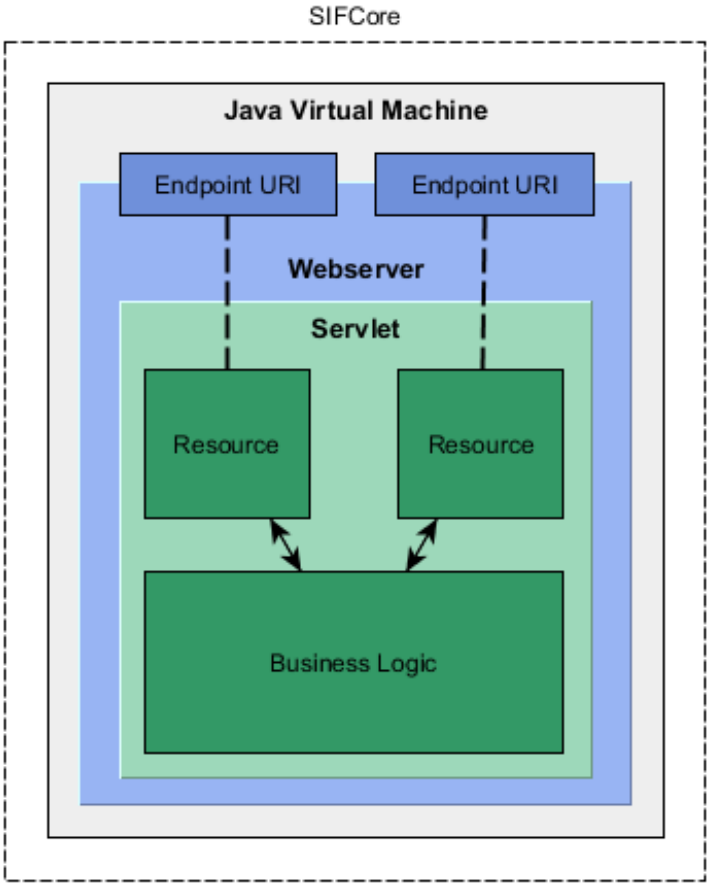

Abbildung 6.1: Komponenten der serviceorientierten Architektur von SIFCore

soll. Die Antwort (engl. Response) liefert dann die Prüfergebnisse und die dazugehörigen Metainformationen zurück an den Client. Die schematische Darstellung dieser Kommunikation wird in Abbildung 6.2 gezeigt.

6.1.3 Dependency Injection

Die Verwendung des Entwurfsmuster Dependency Injection (DI) hat zur Folge, dass es eine zentrale Komponente geben muss, die für die Erstellung der als Abhängigkeit definierten Objekte zur Laufzeit sorgt und diese dann an den benötigten Stellen injiziert. Dazu wird ein sogenannter *Injektor* erstellt, der für die Konfiguration des Abhängigkeitsgraphen zuständig ist. Auf die konkrete Umsetzung der DI wird genauer im Kapitel Implementierung eingegangen. Alle für die DI benötigten Komponenten werden im *App* Modul abgelegt. Ein weiterer Vorteil der DI ist, dass für einzelne Objekte bestimmte Geltungsbereiche festgelegt werden können. So ist es möglich, das Spreadsheet-Modell und die dazugehörigen Schnittstellen an einzelne Anfragen des Webservice zu binden. So kann innerhalb seines Geltungsbereichs leicht auf das Objekt zugegriffen werden, die Objekte einzelner Anfragen überschneiden sich jedoch nicht. Damit ermöglicht die DI, dass SIFCore mehrere Anfragen gleichzeitig und unabhängig bearbeiten kann.

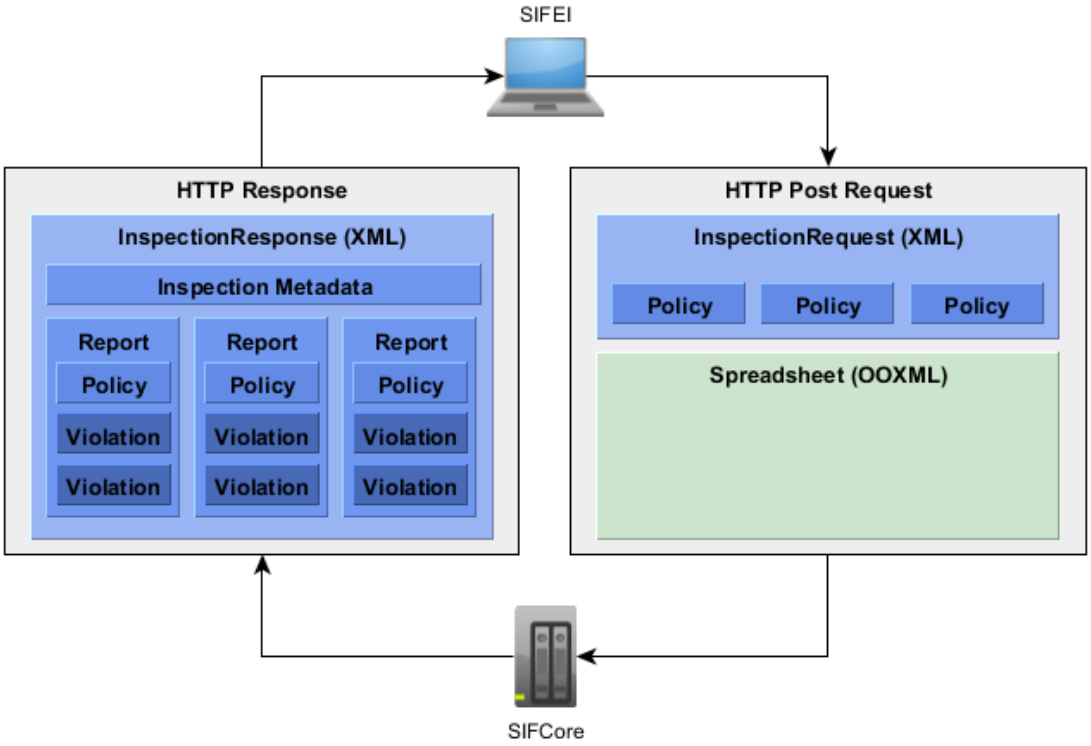

Abbildung 6.2: RESTful Kommunikation von SIFCore

6.2 Metapher

Mit der Einführung der neuen Konzepte und der Restrukturierung der Architektur kann die bisherige zentrale Metapher nicht mehr verwendet werden. Die Nützlichkeit einer neuen Metapher steht in Frage, weil sich im Rahmen der Architekturanalyse die zentrale Metapher als eine der Schwächen der Architektur herausgestellt hat. Meiner Meinung nach unterstützt eine komplexe Metapher, die viele Aspekte der Architektur „erklären“ soll, den Hang von Entwicklern, die Architektur nicht ausreichend zu beschreiben. Aus diesem Grund wird für diese Architektur keine zentrale Metapher definiert, sondern das Verständnis für die Architektur soll mit einer klar strukturierten Architekturbeschreibung gefördert werden. Dies ist keine allgemeine Kritik an dem Prinzip der Metapher zur Architekturbeschreibung, wenn diese sinnvoll ist und das Verständnis der Architektur dadurch verbessert werden kann. So werden insbesondere für die Bezeichnung der Komponenten auch weiterhin Metaphern verwendet, es wird aber auf eine zentrale Metapher verzichtet.

6.3 Struktur von SIFCore

Die Struktur von SIFCore wird an das SOA-Architekturmuster angepasst und besteht aus sieben einzelnen Modulen, die jeweils eine eigene Aufgabe erfüllen. Die Struktur der Module folgt einem

Schichtenmodell, das in Abbildung 6.3 dargestellt wird. Dabei haben die einzelnen Elemente die folgende Bedeutung:

- Mit einer grauen, gestrichelten Linie sind die Schichten der Architektur eingezeichnet.
- Mit einer schwarzen, gepunktet und gestrichelten Linie umrandete Module sind nicht Teil von SIFCore.
- Abhängigkeiten zwischen Modulen sind mit einem gestrichelten Pfeil dargestellt.

Es werden nun die einzelnen Module mit ihren Aufgaben vorgestellt und beschrieben. Für die Notation der Klassendiagramme wird die UML2-Spezifikation verwendet:

- Ein blauer ausgefüllter Pfeil stellt eine Generalisierung dar.
- Ein gestrichelter Pfeil stellt eine Abhängigkeit dar.
- Ein grüner, gestrichelter Pfeil stellt eine spezielle Abhängigkeit in Form einer Schnittstellenrealisierungsbeziehung dar.
- Ein schwarzer Pfeil mit ausgefüllter Raute ist eine Komposition, die Zahlen beziehen sich auf die Multiplizität.

6.3.1 Modul: *App*

Im *App*-Modul (dargestellt in Abbildung 6.4) werden die Konfigurationen des Webservers und des Injektors der DI abgelegt. Für jedes der Module wird hier eine Klasse zum Registrieren der einzelnen Injektionspunkte und Objektfabriken angelegt.

6.3.2 Modul: *API*

Das *API*-Modul (dargestellt in Abbildung 6.5) beinhaltet die *Resources*, die vom Webserver als Endpunkte für die REST-Kommunikation genutzt werden. Dabei existiert für jedes Spreadsheet-Format (Microsoft Excel oder ODF) ein eigener Endpunkt. Zusätzlich befindet sich der *InspectionService* in diesem Modul. Er erteilt den Auftrag zum Erstellen des *SpreadsheetInventory* und übergibt dieses an die Geschäftsprozesse, respektive die *Scanner* und *Facilities*. In 6.5 wird das Klassendiagramm des Moduls in UML-Notation dargestellt.

6.3.3 Modul: *IO*

Das *IO*-Modul (dargestellt in Abbildung 6.6) ist für die Verarbeitung der vom API-Modul weitergereichten *Policies* und *Spreadsheets* zuständig. Für die Spreadsheets existieren verschiedene, nach Spreadsheet-Formaten getrennte Implementierungen der *SpreadsheetIO*-Schnittstelle, die ein Spreadsheet in ein von SIFCore genutztes *Model* umwandeln kann. Die *PolicyIO*-Schnittstelle parst die XML-Datei und erstellt daraus ein *InspectionRequest*.

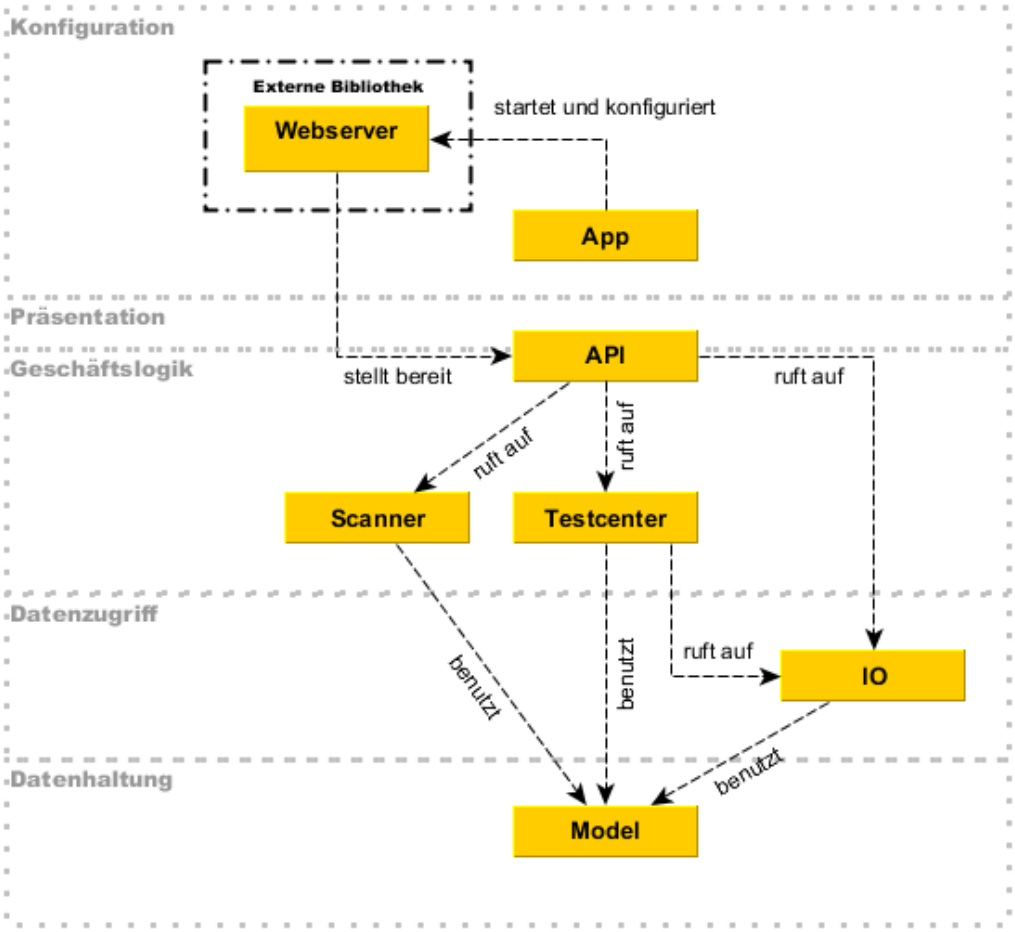

Abbildung 6.3: Module von SIFCore

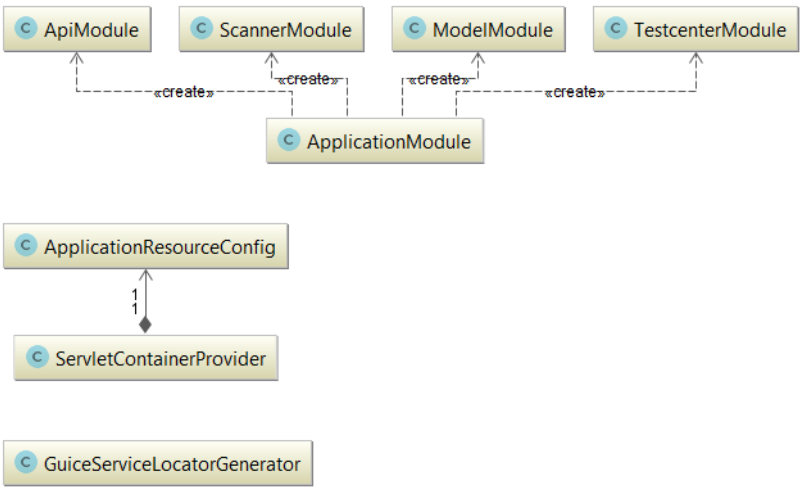

Abbildung 6.4: UML-Klassendiagramm des *App*-Moduls

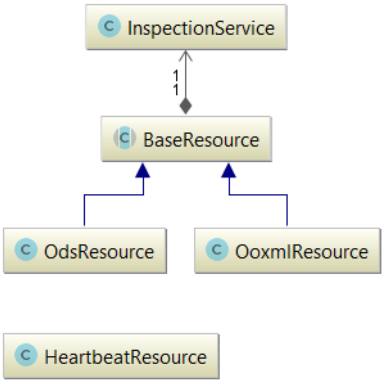

Abbildung 6.5: UML-Klassendiagramm des *API*-Moduls

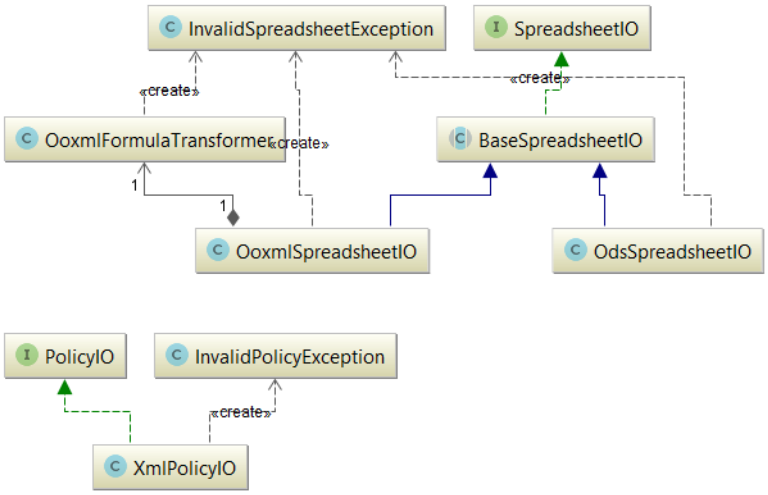

Abbildung 6.6: UML-Klassendiagramm des *IO*-Moduls

6.3.4 Modul: *Model*

Das *Model*-Modul (dargestellt in Abbildung 6.7) beinhaltet alle Elemente, die für die Speicherung eines Spreadsheet-Modells benötigt werden. Die wichtigsten Elemente sind das *Spreadsheet* selbst, *Row*, *Column* und *Cell*.

Ausführbare Funktionen auf Spreadsheets werden *Formula* genannt und sind in den einzelnen Zellen gespeichert. Die Elemente dieser Formeln werden *Token* genannt und in einem eigenen Modul abgelegt. Zusätzlich können einige Elemente verschiedene skalare Werte speichern. Datenhaltungsobjekte für diese Werte werden *Value* genannt und ebenfalls in einem eigenen Modul abgelegt. Aus Gründen der Übersichtlichkeit wurde auf dem in 6.7 gezeigten Klassendiagramm auf die Darstellung der Kompositionen und Abhängigkeiten verzichtet.

Das neue Spreadsheet-Modell orientiert sich dabei offensichtlich stark an dem bisherigen Modell, das in der Arbeit von Sebastian Zitzelsberger [Zit12] beschrieben wird. Bei der Analyse dieses

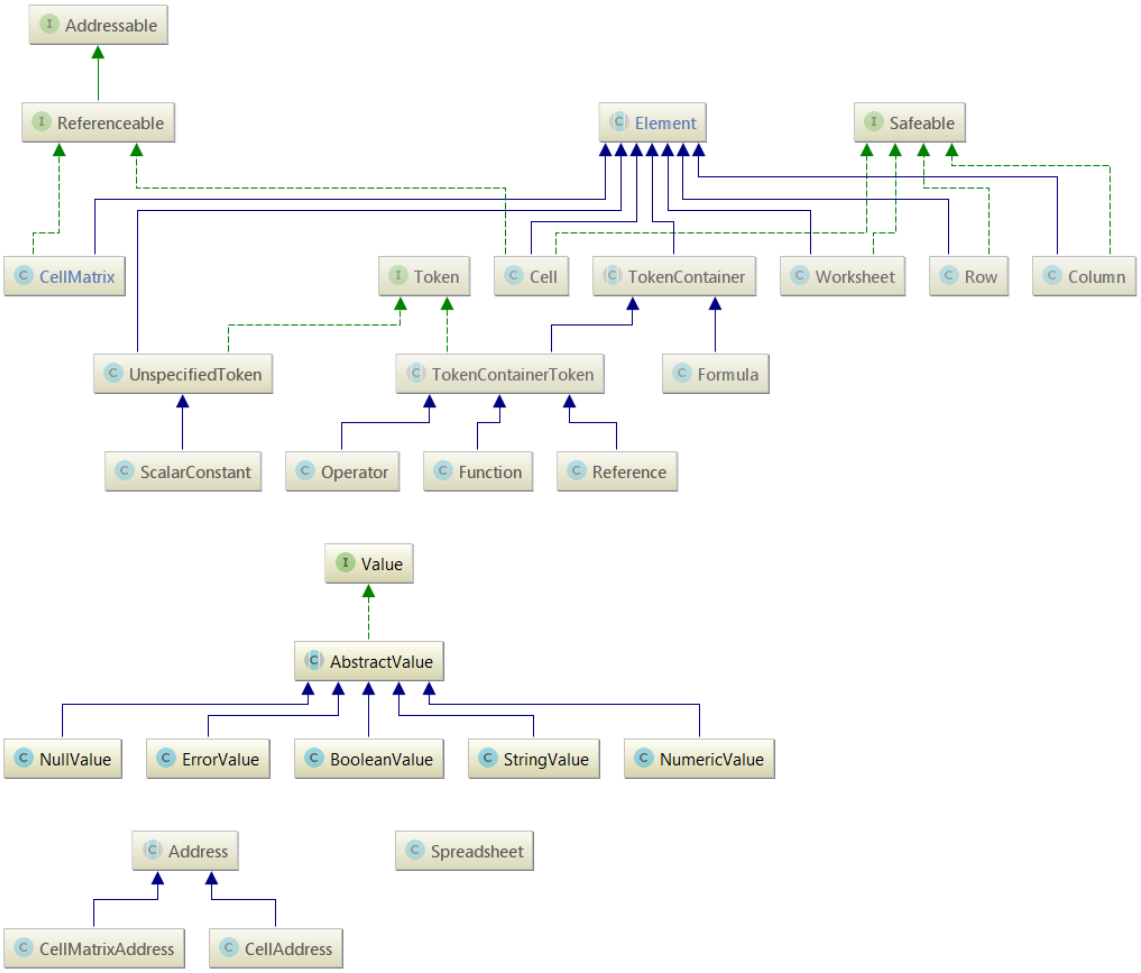

Abbildung 6.7: UML-Klassendiagramm des *Model*-Moduls

Modells wurden jedoch einige Probleme festgestellt, die im Kapitel 4 dokumentiert wurden. Aus diesem Grund werden einige Veränderungen und Vereinfachungen vorgenommen. So werden die Markierungs-Schnittstellen entfernt, also alle Schnittstellen die keine Funktionalität anbieten. Außerdem wird die Vererbungs-Hierarchie vereinfacht und nicht benötigte Vererbungen entfernt. Auch die Benennung der Schnittstellen und Klassen wird geändert, sodass diese einheitlich und besser verständlich ist.

6.3.5 Modul: *Testcenter*

Das *Testcenter*-Modul (dargestellt in Abbildung 6.8) beinhaltet alle Komponenten die für eine Prüfung benötigt werden. Die einzelnen Prüfungen werden dabei jeweils eigenen Modulen zugeordnet, erweitern aber die Basisklassen des *Testcenter*-Moduls. Die zentrale Komponente ist die Basisklasse aller *Facilities*, die *AbstractFacility*, über die auf das *SpreadsheetInventory* zugegriffen werden kann. Diese beinhaltet sowohl den *InspectionRequest* mit den verwendeten *Policies*, als

auch die *InspectionResponse* mit den gemeldeten *Violations*. Die Prüfung des Spreadsheets ist die zentrale Funktionalität von SIFCore, aus diesem Grund werden später in diesem Kapitel, die einzelnen Komponenten und der Ablauf einer solchen Prüfung noch detaillierter beschrieben.

Im dargestellten UML-Diagramm wird aus Gründen der Übersicht nur das für die Prüfung „Gleiche Verweise“ benötigte Modul eingezeichnet, die Abhängigkeiten sind aber für alle anderen Prüfungen gleich.

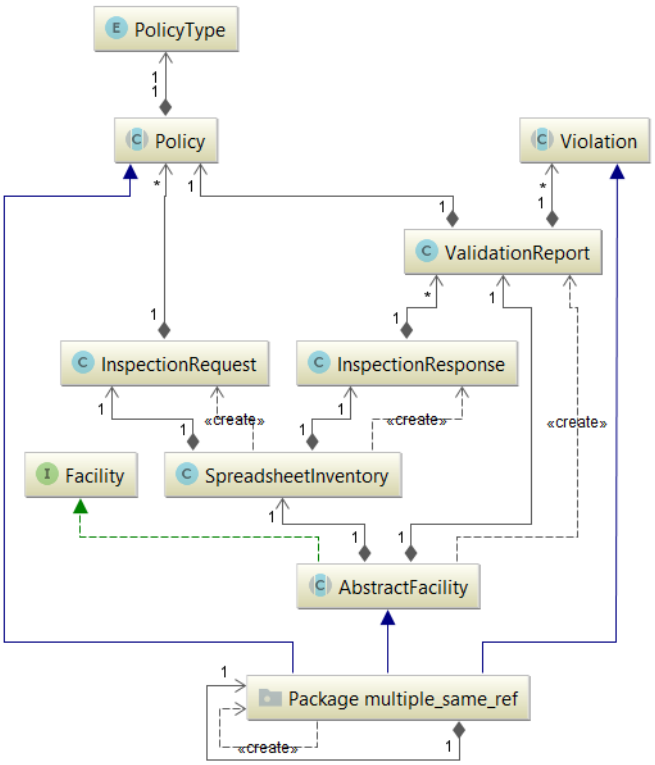

Abbildung 6.8: UML-Klassendiagramm des *Testcenter*-Moduls

6.3.6 Modul: *Scanner*

Das *Scanner*-Modul (dargestellt in Abbildung 6.9) beinhaltet alle Komponenten, die für die Analyse des internen Spreadsheet-Modells benötigt werden. Diese Komponenten implementieren die *Scanner*-Schnittstelle und können verschiedene, vom Entwickler des jeweiligen *Scanners* definierte, Elemente in einer Liste im *SpreadsheetInventory* speichern und so allen Prüfungen zur Verfügung stellen. Die *Scanner* werden dabei nach dem Erstellen des Spreadsheet-Modells, aber vor der Ausführung der Prüfungen gestartet.

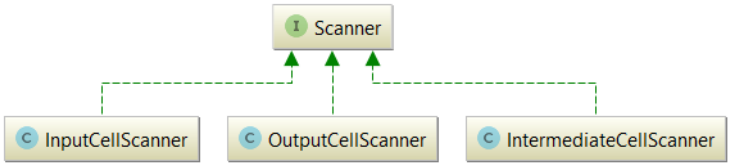

Abbildung 6.9: UML-Klassendiagramm des *Scanner*-Moduls

6.3.7 Modul: *Utility*

Im *Utility*-Modul (dargestellt in Abbildung 6.10) befinden sich kleinere Komponenten, die an verschiedenen Stellen der Software benötigt werden, aber keinem anderen Modul sinnvoll zugeordnet werden können. Dazu gehören die Komponenten zur Konvertierung von Werten und die Übersetzungskomponente.

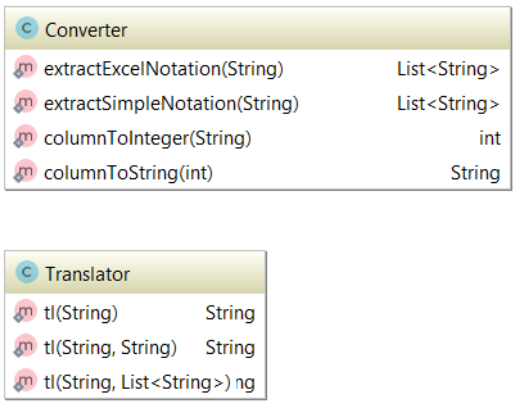

Abbildung 6.10: UML-Klassendiagramm des *Utility*-Moduls

6.4 Spreadsheet-Prüfung

Die Spreadsheet-Prüfung ist die zentrale Funktionalität, die SIFCore per Webservice anbietet. In diesem Abschnitt werden die einzelnen, an einer Prüfung beteiligten, Komponenten und der Ablauf einer solchen Prüfung beschrieben.

6.4.1 Komponenten der Spreadsheet-Prüfung

In der Arbeit von Sebastian Zitzelsberger [Zit12] findet sich im Kapitel „Konzept“ eine Beschreibung der Komponenten, die für die erste Umsetzung von SIFCore verwendet wurden. Die Komponenten der neuen Architektur orientieren sich, soweit dies durch die Entfernung der bisherigen Metapher und die Einführung der neuen Konzepte SOA und DI möglich ist, an dieser

Beschreibung. Weil die Implementierung von SIFCore in englischer Sprache umgesetzt wurde und dies auch weiterhin so gehandhabt werden soll, werden die Komponenten mit ihren englischen Bezeichnern vorgestellt. So sollen Verständnisprobleme vermieden werden, die entstehen wenn Komponenten mit deutschen Bezeichnern entworfen, aber mit englischen Bezeichnern implementiert werden.

- **SpreadsheetInventory**: Diese Komponente speichert das *Spreadsheet*, den *InspectionRequest*, die *InspectionResponse* und die von den Scannern erstellen Listen von Elementen.
- **Spreadsheet**: Diese Komponente repräsentiert das Spreadsheet-Modell und es können damit Zeilen, Spalten, Zellen, Werte oder Formeln des Spreadsheets abgerufen werden.
- **InspectionRequest**: Diese Komponente repräsentiert das XML-Dokument, das vom Benutzer des Webservice gesendet wurde und beinhaltet somit die *Policies*.
- **InspectionResponse**: Diese Komponente repräsentiert das XML-Dokument, das als Antwort an den Benutzer des Webservice gesendet wird und beinhaltet somit die *Violations*.
- **Policy**: Diese Komponente wird für Konfiguration einer *Facility* verwendet. Es existiert für jede Facility eine eigene Policy, die alle benötigten Konfigurationsparameter der Prüfung beinhaltet.
- **Violation**: Diese Komponente wird für die Speichern eines gefunden Verstoßes bei der Prüfung des Spreadsheets verwendet. Alle *Violations*, die aufgrund einer das Policy gefunden wurden, werden in einem *ValidationReport* zusammengefasst.
- **Facility**: Dies ist die ausführbare Komponente einer Spreadsheet-Prüfung und hier wird die Funktionalität der einzelnen Prüfungen implementiert. In die *Facility* wird das *SpreadsheetInventory* injiziert und so kann die Konfiguration der Prüfung über die zugehörige *Policy* vorgenommen werden, das Spreadsheet-Modell untersucht und ein gefundenes Problem als *Violation* gespeichert werden.

6.4.2 Ablauf einer Spreadsheet-Prüfung

Der Ablauf einer Prüfung wird in Abbildung 6.11 beschrieben. Als Grundlage wird der von Sebastian Zitzelsberger [Zit12] in Kapitel 6.2 beschriebene Ablauf verwendet, dieser wird aber an die neue Architektur angepasst. SIFCore ist als RESTful Webservice ausgelegt, deshalb wird eine Prüfung über eine HTTP-Anfrage gestartet, die der Visualisierungsclient SIFEI erstellt. Die Anfrage wird, über eine der *Resource*-Schnittstellen im API-Modul, entgegen genommen und dann mit Hilfe des *IO*-Moduls weiterverarbeitet. Es wird zuerst das mitgesendete XML-Dokument geparst und damit werden die *Policies* erstellt. Anschließend wird das Spreadsheet importiert und damit das Spreadsheet-Modell erstellt. Dann werden alle registrierten *Scanner* ausgeführt und die verfügbaren Metainformationen im *SpreadsheetInventory* gespeichert. Anschließend werden alle Factories ausgeführt, für die eine Policy geparst wurde. Mit Hilfe der in der *Policy* enthaltenen Konfigurationsdaten, werden bei der Prüfung *Violations* für alle Regelverstöße erstellt. Es werden zum Schluss alle *Violations* in ein XML-Dokument geparst und dieses wird in der HTTP-Antwort

an den Visualisierungsclient zurück geschickt. Dieser kann dann die Verstöße visualisieren und den Benutzer so über die Ergebnisse der Prüfung informieren.

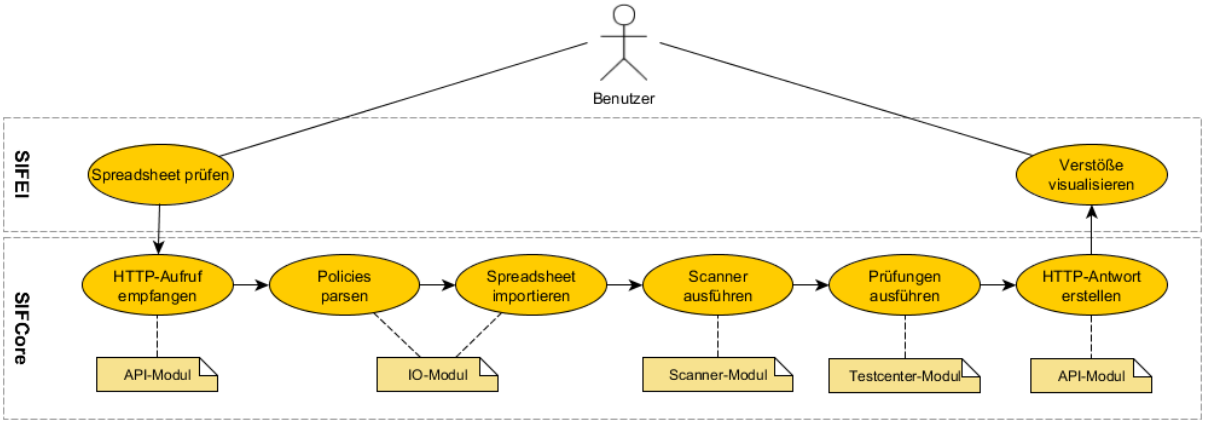

Abbildung 6.11: Ablauf einer Prüfung mit SIFEI und SIFCore

7 Implementierung

Dieses Kapitel behandelt die Umsetzung des Architekturentwurfs und soll es späteren Entwicklern der Software erleichtern, die Software zu warten. Weil diese Abschlussarbeit keine reine Forschungsarbeit ist, sondern auch das sekundäre Ziel hat, die Qualitätsmerkmale von SIFCore sowie einige funktionale Aspekte der Software zu verbessern, dokumentiert dieses Kapitel auch die diesbezüglich vorgenommenen Veränderungen. Dieses Kapitel kann übersprungen werden, wenn kein Interesse an den Implementierungsdetails von SIFCore und SIFEI besteht.

7.1 SIFCore

Da sich die Architektur von SIFCore grundlegend geändert hat, habe ich mich dazu entschieden, auch bei der Implementierung einen klaren Schnitt zu machen. Aus diesem Grund wurde zuerst das Grundgerüst von SIFCore auf Basis des Entwurfs neu implementiert. Anschließend wurden die bisherigen Komponenten, soweit benötigt, einzeln überarbeitet und integriert.

Diese Gelegenheit wurde auch genutzt um das *Gradle* genannte Build-Management und Automatisierungs-Tool zu integrieren. Bisher wurden die externen Java-Bibliotheken manuell in das Projektarchiv kopiert, nun ist es möglich über die Datei *gradle.build* im Hauptverzeichnis die benötigten Abhängigkeiten mit expliziter Angabe der gewünschten Version zu definieren. Die Konfiguration von *Gradle* und der verwendeten Abhängigkeiten wird dabei über die Datei *build.gradle* im Projektverzeichnis vorgenommen.

Mit Hilfe von *Gradle* ist auch wesentlich einfacher ein Deployment durchzuführen. So können alle Abhängigkeiten von einem *Maven*-Repository während des Buildvorgangs (aus dem Internet oder einem Offline-Repository) heruntergeladen werden. Dies stellt sicher, dass auch Abhängigkeiten die nicht im SIFCore-Projektarchiv enthalten sind, mit der gewünschten Version verwendet werden. Auch können Modul- und Integrationstests automatisiert ausgeführt werden und eine Anbindung an einen *continuous integration service* wie etwa *Jenkins* ist sehr einfach, was eine Entwicklung nach dem Prinzip der kontinuierlichen Integration ermöglicht. Dieses Vorgehen stellte auch sicher, dass mit der Integration immer weiterer Komponenten aus den alten SIFCore-Versionen keine Fehler entstehen, die durch Nebeneffekte ausgelöst werden.

7.1.1 RESTful Webservice

Für die Umsetzung des Webservice existiert die Java-Spezifikation JAX-RS, die eine Schnittstelle spezifiziert, die die Verwendung des Software-Architekturstils Representational State Transfer

(REST) im Rahmen von Webservices ermöglicht und vereinheitlicht. Für diese Spezifikation existiert auch eine Referenzimplementierung unter dem Namen *Jersey* der Firma *Oracle*, die in SIFCore integriert wird.

Die Endpunkte des Webservice werden über die von *Jersey* bereitgestellten Annotationen in den *Resource*-Klassen, die sich im *API*-Modul befinden, definiert. Das XML-Dokument für die *Policies* und das eigentliche Spreadsheet müssen dabei als Datenstrom vom Typ *multipart/form-data* in einer einzigen HTTP-Anfrage an SIFCore gesendet werden. Dieser Datenstrom wird in den Endpunkten aufgetrennt und über die *IO*-Schnittstellen in die gewünschten Objekte umgewandelt. Anschließend wird mit dem *InspectionService* die Geschäftslogik gestartet und es werden die *Scanner* und *Facilites* aufgerufen. Zum Schluss erstellt der *InspectionService* das XML-Dokument, das die *Violations* und Metadaten der Prüfungen enthält. Dieses Dokument wird dann vom Endpunkt mittels einer HTTP-Antwort zurück an den Benutzer des Webservices gesendet.

7.1.2 Dependency Injection Framework

Für die Umsetzung der DI wird das *Google Guice*-Framework genutzt. Dazu wird in der *main*-Methode von SIFCore ein sogenannter *Injektor* geladen. Dieser Injektor kann die mit Annotationen gekennzeichneten Injektionspunkte finden und dann das gewünschte Objekt injizieren. Damit das Objekt injiziert werden kann, muss es zuvor noch an den Injektor *gebunden* werden. Für eine bessere Übersicht gliedert Guice die Bindungen und anderen Konfigurationen in sogenannte Module. Auch für SIFCore wurden verschiedene Module angelegt, um eine bessere Übersicht zu gewährleisten. Es existieren die folgenden Guice-Module, die sich alle im Java-Package *sif.modules* befinden:

- **ApplicationModule**: Das erste Modul das geladen wird, beinhaltet die Verweise auf andere Guice-Module und die Konfiguration des Guice-Servlet-Containers.
- **ApiModule**: Bindet die REST-*Resources* und die *SpreadsheetIO*-Schnittstelle an ihre Implementierungen.
- **ScannerModule**: Bindet die verfügbaren *Scanner* mit Hilfe eines Multibinders. Mit diesem können alle gebunden Objekte als *Set* gleichzeitig geladen werden.
- **TestcenterModule**: Bindet die verfügbaren *Facilities* ebenfalls mit Hilfe eines Multibinders.
- **ModelModule**: Bindet die Spreadsheet-Komponenten und stellt *Factories* bereit, mit denen diese erstellt werden können.

Für weitere Details wie die DI in der Implementierung umgesetzt wird, wird auf das exzellente *Google Guice Wiki*[1] verwiesen. Es wurde bei der Implementierung darauf geachtet, dass alle *Best Practices* die von den Guice-Entwicklern empfohlen werden, auch umgesetzt wurden.

[1] https://github.com/google/guice/wiki

7.1.3 Erstellung des Spreadsheet-Modells

Die Erstellung des Spreadsheet-Modells wird über die Schnittstelle *SpreadsheetIO* umgesetzt. Die Implementierung dieser Schnittstelle wurde im Zuge der Integration in die neue Architektur überarbeitet. Die Methoden, Bedingungen und Schleifen in den Klassen *OoxmlSpreadsheetIO* (für das gesamte Spreadsheet) und *OoxmlFormulaTransformer* (für die Formeln im Spreadsheet) wurden so bearbeitet, dass ihre Komplexität und Größe deutlich reduziert werden konnte. Dies wurde durch eine Aufsplittung von Funktionalität in neue Methoden und die Wiederverwendung dieser Methoden erreicht.

7.1.4 SpRuDeL XML-Format

Das SpRuDeL XML-Format wurde in der zweiten Entwicklungsstufe ([Lem13]) von SIFCore eingeführt und dient der Konfiguration der Spreadsheet-Prüfung. Die Anpassungen des Spreadsheet-Modells ziehen dabei auch einige Veränderungen des Austauschformats nach sich. So wurde die alte *DynamicPolicy* entfernt und es werden nun zwei getrennte Austauschobjekte für die Anfrage, beziehungsweise den Inspektionswunsch (*InspectionRequest*), und die Antwort (*InspectionResponse*) verwendet.

Der *InspectionRequest* beinhaltet eine Liste von *Policies*, die während der Prüfung des Spreadsheets angewendet werden sollen sowie die von der Prüfung global ignorierten Zellen und Arbeitsblätter. Jede *Policy* besitzt dabei mindestens die von der Basisklasse definierten Felder, kann dabei jedoch um weitere beliebig komplexe Datenobjekte erweitert werden, wenn die damit verknüpfte Prüfung diese für die Ausführung benötigt. So werden zum Beispiel bei der *DynamicTesting-Policy*, die als Austauschobjekt für die *DynamicTestingFacility* dient, die Daten der Szenarien mit einer Liste von *Scenario*-Objekten übergeben. Diese Objekte enthalten wiederum Listen von *Input*-, *Invariant*- und *Output*-Objekten, die dann die Bedingungen für einzelne Zellen auf dem Spreadsheet enthalten.

Die *InspectionResponse* wird nun als Austauschobjekt für die Antwort auf eine Prüfungsanfrage verwendet. Sie enthält dabei eine Liste von *ValidationReports*, einen für jede ausgeführte Prüfung. Dieser *ValidationReport* enthält die für die Prüfung verwendete *Policy* und eine Liste von *Violations*, die während der Prüfung gefunden wurden. Des weiteren werden Fehler oder Probleme, die während der Prüfungen aufgetreten sind, in einer Liste in der *InspectionResponse* gespeichert.

Um die Korrektheit der XML-Dokumente in Client-Anwendungen überprüfen zu können, wurden außerdem mit Hilfe des JAXB XML-Parser für beide Austauschobjekte XML-Schema-Definitionen (XSD) erstellt und im Paket *sif.testcenter* hinterlegt. Bis jetzt existiert jedoch keine Möglichkeit diese automatisiert, zum Beispiel über einen eigenen Endpunkt im Webservice abrufen zu können, sodass sie manuell zur Verfügung gestellt werden müssen.

7.1.5 Integration der Prüfungen

Die Integration der Prüfungen wurde schrittweise nach der Implementierung des Grundgerüsts vollzogen. Dabei wurden für jede Prüfung die drei elementaren Klassen (*Facility*, *Policy* und *Violation*) aus der bisherigen SIFCore-Version extrahiert und in ein neues Java-Paket im *Testingcenter* verschoben. Anschließend wurden diese Klassen überarbeitet und an die neue Architektur angepasst.

Um eine *Policy* mit der entsprechenden *Facility* zu verknüpfen, müssen die Methoden *getPolicyClass()* und *setPolicy()* in der *Facility* überladen werden. So kann jede *Facility* selbständig entscheiden ob im *InspectionRequest* eine für sie bestimmte *Policy* enthalten ist. Der *InspectionService* ruft dafür für jede Facility die *runCheck()*-Methode auf und prüft so, ob eine entsprechende Policy existiert. Die Funktionalität der Prüfung wird dann in der *run()*-Methode umgesetzt, die aufgerufen wird, wenn die *runCheck()*-Methode erfolgreich war. In dieser *run()*-Methode werden dann auch die einzelnen *Violations* erstellt und dann im Feld *ValidationReport* gespeichert. Nach der Prüfung wird dieser *ValidationReport* dann der *InspectionResponse* übergeben.

Bei der Überarbeitung der Prüfungen wurden, neben der Anpassungen an die neue Architektur, außerdem noch weitere Veränderungen vorgenommen. So wurde die Komplexität der Methoden reduziert, indem diese aufgesplittet wurden. Auch wurden Bedingungsanweisungen und Schleifen entzerrt und Fehlerkontrollmechanismen eingefügt. Durch diese Maßnahmen konnten viele, durch die Analyse in diesen Klassen festgestellten Probleme gelöst und insbesondere die Existenz von doppelt vorhandenem Quellcode deutlich reduziert werden.

7.1.6 Internationalisierbarkeit

Die bisherige Umsetzung der Internationalisierbarkeit und damit hauptsächlich die Übersetzung der gemeldeten *Violations* war sehr umständlich und es konnten nur einzelne statische Zeichenketten übersetzt werden. Die bisher für die Übersetzung zuständige Klasse *Translator* wurde dafür aus der vorherigen Implementierung übernommen und erweitert. Anstatt das Singleton-Pattern zu implementieren und eine einzige Übersetzungsfunktion anzubieten, gibt es in dieser Klasse jetzt verschiedene statische Funktionen für die Übersetzung mit dynamischem Inhalt. So ist es nun möglich Variablen vom Typ *string*, einzeln oder als Liste, als Parameter für die Übersetzungsfunktion zu verwenden, die dann an Stelle des Platzhalters in die Ausgabe eingefügt werden.

Weil die verwendete Java-Implementierung (Java 8) zur Zeit keine Unterstützung für UTF-8 bei sogenannten *Resource Bundles* bietet, auf denen die Übersetzung basiert, wird diese Unterstützung mit einem Workaround ermöglicht. Dieser ist im Quellcode gekennzeichnet, falls zukünftige Java-Versionen diesen nicht mehr benötigen.

7.1.7 Tests

Um die Funktionalität von SIFCore überprüfen zu können und sicher zu stellen, dass die Implementierung mit der Spezifikation übereinstimmt wurden verschiedene Tests erstellt, die in Modul- und Integrationstests unterteilt werden. Um die Aussagekraft und Vollständigkeit der Tests einschätzen zu können, werden die von den Tests abgedeckten Klassen, Methoden oder Quellcodezeilen erfasst und mit den jeweiligen Gesamtzahlen verglichen. In der Spezifikation von SIFCore in Kapitel 6 wird eine Testabdeckung von mindestens 80% gefordert. Dieser Wert wurde für die getesteten Elemente auch erreicht oder übertroffen, wie der in Tabelle 7.1 dargestellte Testreport beweist. Die Testabdeckung der Modul- und Integrationstests wird dabei mit dem Werkzeug IntelliJ IDEA erfasst und es werden die Abdeckungen beider Testtypen zusammengefasst.

Element	Klassen	Methoden	Zeilen
sif.api	100% (5/5)	100% (14/14)	100% (74/74)
sif.app	100% (9/9)	100% (12/12)	100% (73/73)
sif.io	100% (10/10)	92% (36/39)	80% (235/292)
sif.model	96% (31/32)	71% (150/211)	74% (403/542)
sif.scanner	100% (3/3)	100% (3/3)	100% (30/30)
sif.testcenter	95% (46/48)	89% (232/260)	90% (842/928)
sif.utility	100% (2/2)	100% (8/8)	89% (43/48)

Tabelle 7.1: Testabdeckung von SIFCore

Modultests

Um die Korrektheit von einzelnen Modulen oder Klassen sicherzustellen, wurden für SIFCore verschiedene Modultests implementiert. Dabei wird auf das weit verbreitete Testframework JUnit zurückgegriffen. In der aktuellen Version von SIFCore wird die Erstellung der XML-Dokumente (*siftest.xml.XMLTest*), die Konvertierung von Einheiten (*siftest.utility.ConverterTest*) und die Übersetzungsfunktion (*siftest.utility.TranslatorTest*) einzeln getestet. Die Funktionalität von Prüfungen wird hingegen, auf Grund der hohen Anzahl an beteiligten Modulen und Klassen, von Integrationstest überprüft, die im nächsten Abschnitt beschrieben werden.

Integrationstests

Die Integrationstest von SIFCore sollen sicherstellen, dass die Funktionalität von SIFCore vollständig und korrekt ist. Die Integrationstest prüfen dabei den gesamten Ablauf einer Spreadsheet-Prüfung, von der Entgegennahme der HTTP-Anfrage, über die Prüfung des Spreadsheets bis hin zur HTTP-Antwort. Für jede einzelne *Facility* wird ein separater Integrationstest implementiert.

Zusätzlich werden noch zwei Integrationstests umgesetzt, die alle verfügbaren Prüfungen gleichzeitig durchführen können. Es wird dabei ein Test für jedes der beiden unterstützen Microsoft Excel-Formate (*xls* bekannt als EXCEL97 und *xlsx* bekannt als OOXML) erstellt.

Die Integrationstests sind im Java-Paket *siftest.zzz_integration_test* abgelegt und werden auch in Form von JUnit-Tests implementiert. Dabei wird im Unterschied zu den Modultests jedoch nicht nur eine einzige Klasse getestet, sondern es wird mit Hilfe der Jetty-Bibliotheken ein dedizierter *Servlet-Container* erstellt und gestartet, in dem dann das SIFCore-*Servlet* ausgeführt wird. Zusätzlich wird dann ein HTTP-Client (nach JAX-RS Spezifikation) erstellt, der eine Verbindung zum *Servlet-Container* aufbaut. Über diese Verbindung werden dem Server dann die, für den jeweiligen Test angepassten, *Policies* und das *Spreadsheet* übergeben. Nach der Verarbeitung der Anfrage durch das SIFCore-*Servlet*, wird die generierte Antwort mit der für diesen Test erstellten Referenzantwort verglichen und so festgestellt, ob das Verhalten von SIFCore mit der Spezifikation übereinstimmt.

7.2 SIFEI

Nach der Überarbeitung der Kommunikationskomponente von SIFCore, musste auch der Visualisierungsclient SIFEI entsprechende angepasst werden. Weil die Untersuchung der Architektur von SIFEI nicht Teil dieser Arbeit ist, wird die Software nur soweit geändert, dass eine Benutzung wieder möglich ist. Zu diesem Zweck wurden alle alten Kommunikationskomponenten entfernt und die *WorkbookModel* genannte Datenhaltungskomponente so angepasst, dass eine globale und konsistente Bennenung der Zelladressen möglich ist. Es wurde dann mit Hilfe der .NET-Bibliothek ein HTTP-Client erstellt, der mit SIFCore kommunizieren kann. Für eine erfolgreiche Kommunikation musste dann nur noch das verwendete XML-Austauschformat angepasst werden und die neue von SIFCore verwendete SpRuDeL-Version auch in SIFEI implementiert werden.

Bei der Implementierung dieser SIFEI-Komponenten wurden außerdem gleich noch alle offentlichtlichen Fehler und Probleme die mir aufgefallen sind, gleich verbessert oder für eine spätere Überarbeitung durch andere Entwickler markiert. Es wurden einige nicht mehr genutzten Komponenten entfernt, einige Code-Duplikate überarbeitet und die Darstellung der gefundenen *Violations* vereinheitlicht.

8 Architekturanalyse II

In diesem Kapitel werden nun die in Kapitel 4 durchgeführten Analyseverfahren erneut genutzt, um die verbesserte Architektur von SIFCore zu bewerten. Die Gliederung und der Ablauf der Analyse wird dabei nicht geändert, sodass in diesem Kapitel nur noch die Ergebnisse dieser zweiten Analyse vorgestellt werden.

8.1 Szenarien

Zuerst werden die beiden szenario-basierten Verfahren SAAM und ALMA für die neue Architektur durchgeführt. Dabei werden dementsprechend auch die gleichen Szenarien wie in Kapitel 4 verwendet. Auf eine weitere Beschreibung der Details zur Durchführung wird verzichtet und es werden direkt die Ergebnisse der Verfahren vorgestellt. Wie für die frühere Architektur von SIFCore werden die Ergebnisse der Auswertung der Szenarien von SAAM und ALMA in einem Schritt zusammengefasst.

Szenario 1: Erstellen einer neuen Prüfung

Als Erstes wird im Modul *testcenter* ein neues Modul für eine Prüfung erstellt. Die Prüfung von Spreadsheets wird weiterhin in den *Facilities* umgesetzt. Dazu wird eine neue Komponente erstellt, die eine Spezialisierung der *testcenter.AbstractFacility* ist. Für die Konfiguration der benötigten Parameter wird eine neue *Policy*, die eine Spezialisierung von *testcenter.Policy* ist, angelegt. Es wird mindestens eine neue Klasse, abgeleitet von *testcenter.Violation*, erstellt und als Datenaustauschobjekt für die Verstöße der Prüfungen verwendet. All diese Komponenten werden dem neu erstellten Modul zugeordnet. Damit die Prüfung nun bei einer entsprechenden *Policy* auch aufgerufen wird, muss die *Facility* noch im *app.TestcenterModule* registriert werden und die dazugehörige *Policy* muss im *testcenter.InspectionRequest* referenziert werden.

Über die Schnittstelle *SpreadsheetIO* kann das Spreadsheet mit neuen Inhalten aus dem Modell aktualisiert werden. Zusätzlich können so Funktionen auf dem Spreadsheet ausgeführt werden und die Ergebnisse davon im Modell gespeichert werden.

Szenario 2: Unterstützung für ein neues Spreadsheet-Format

Um ein neues Spreadsheet-Format zu unterstützen, muss die Schnittstelle *SpreadsheetIO*, die den direkten Zugriff auf das Spreadsheet ermöglicht, für die neue Bibliothek die das entsprechende Spreadsheet-Format lesen kann, implementiert werden. Zusätzlich muss ein neuer Endpunkt des Webservice erstellt werden, bei dem Prüfungen für das neue Spreadsheet-Format durchgeführt werden können. Dieser Endpunkt muss die neue Implementierung der *SpreadsheetIO*-Schnittstelle injizieren und kann ansonsten, äquivalent zu den bisherigen Endpunkten, den *InspectionService* nutzen, um Prüfungen durchzuführen.

8.2 Metriken

Analog zum Vorgehen bei der ersten Architekturanalyse werden zuerst die einzelnen, ausgesuchten Metriken betrachtet, dann die Pseudometriken der beiden Analysewerkzeuge und zum Schluss wird eine DSM-Analyse durchgeführt.

8.2.1 Einzelmetriken

Die Anzahl an Java-Packages (26), Klassen (131) und Funktionen (623) wird erneut durch Sonarqube bestimmt und in Tabelle 8.1 dargestellt.

8.2.2 Pseudometriken

Die Bewertung durch die Werkzeuge Sonarqube und Teamscale wird mit der gleichen Konfiguration wie bei der ersten Analyse durchgeführt.

Ergebnisse der Bewertung durch Teamscale

Die erneute Bewertung durch Teamscale generiert die in Abbildung 8.1 dargestellte Übersicht. In Abbildung 8.2 werden erneut die gefunden Probleme nach Fehlergrad und Art des Fehlers aufgeschlüsselt.

Ergebnisse der Bewertung durch Sonarqube

Die erneute Bewertung durch Sonarqube generiert die in Abbildung 8.3 dargestellte Übersicht.

8.2.3 DSM

Die in Abbildung 8.4 dargestellte DSM wurde erneut mit Hilfe von IntelliJ IDEA erstellt.

Metrik	Wert absolut	Wert relativ
Module mit vielen Komponenten	0	0%
Module mit wenigen Komponenten	14	53%
Maximale Verschachtelungstiefe von Modulen	7	-
Übermäßig gekoppelte Klassen	0	0%
Klassen mit vielen Methoden	1	0,8%
Verkettungen von *"instance of"* Prüfungen	0	-
Markierungsschnittstellen	0	-
Singleton-Klassen	0	0%
Übermäßig gekoppelte Methoden	2	0,3%
Übermäßig verschachtelte Methoden	2	0,3%
Methoden mit konstantem Rückgabewert	2	0,3%
Verworfene Rückgabewerte von Methoden	0	-
Nicht verwendete Klassen	0	0%
Nicht verwendete Methoden	5	0,8%
Stille Exceptions	0	-
Anweisungen mit konstanten Bedingungen	0	-
Ungenutzte Zuweisungen	0	-
Tippfehler	0	-

Tabelle 8.1: Werte der betrachteten Einzelmetriken nach dem Reengineering

8.3 Simulationen und Experimente

Weil für die verbesserte Architektur bisher keine Prototypen veröffentlicht wurden, gibt es auch keine Einträge im Issue-Tracker und eine Analyse entfällt.

8.4 Durchsicht

Die Ergebnisse der Durchsicht der neu erstellten Architekturdokumentation fließt wieder direkt in die Bewertung und den Vergleich der Architekturen ein.

8.5 Bewertung und Vergleich der Architekturen

Für die Bewertung der neuen Architektur von SIFCore wird diese mit der ersten Architektur verglichen und dabei die Verbesserungen herausgearbeitet. Der Vergleich stützt sich dabei auf die in Kapitel 4 und Kapitel 8 durchgeführten Analyseverfahren. Es wird nun auf die in Abschnitt 4.6 formulierten Maßnahmen einzeln eingegangen und die Ergebnisse der Analysemethoden dahingehend interpretiert, ob das Ziel erreicht wurde und die Maßnahme erfolgreich war.

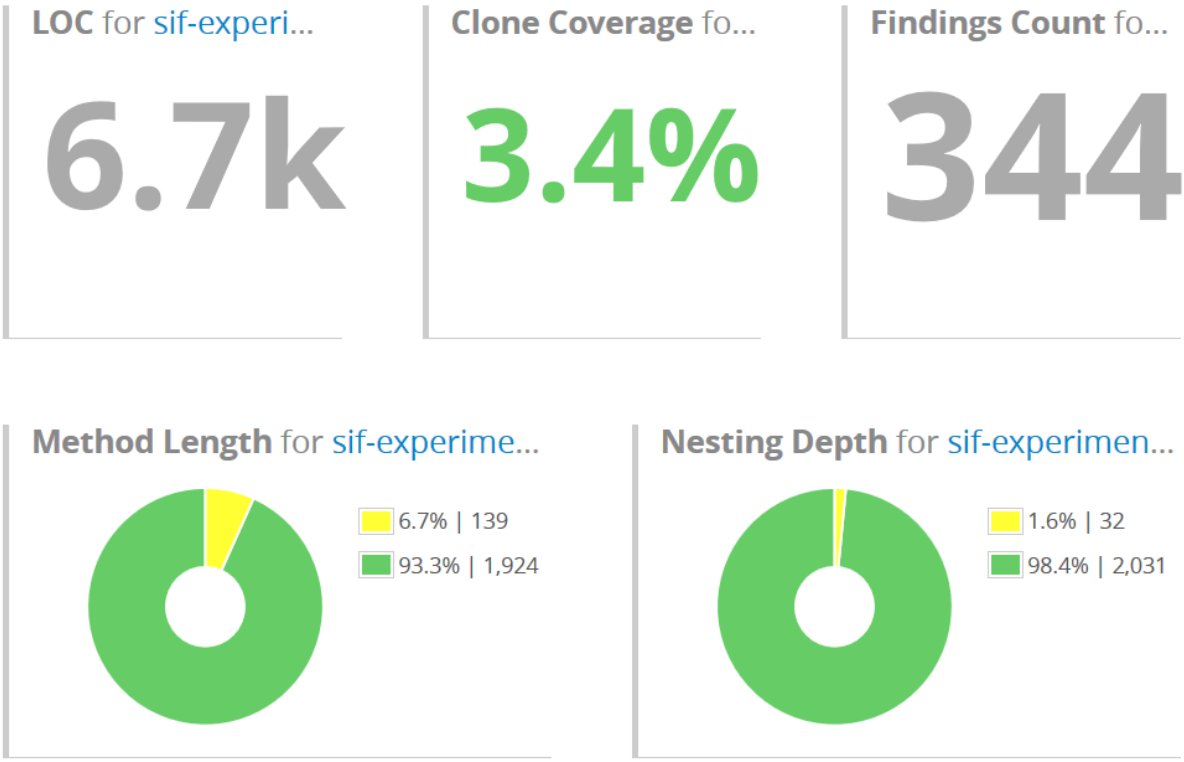

Abbildung 8.1: Das Dashboard von Teamscale für SIFCore nach dem Reengineering

Restrukturierung der Module

Die Restrukturierung der Module war die erste Maßnahme, die zur Verbesserung der Softwarequalität festgelegt wurde. Denn sowohl die szenariobasierten Analyseverfahren, als auch die

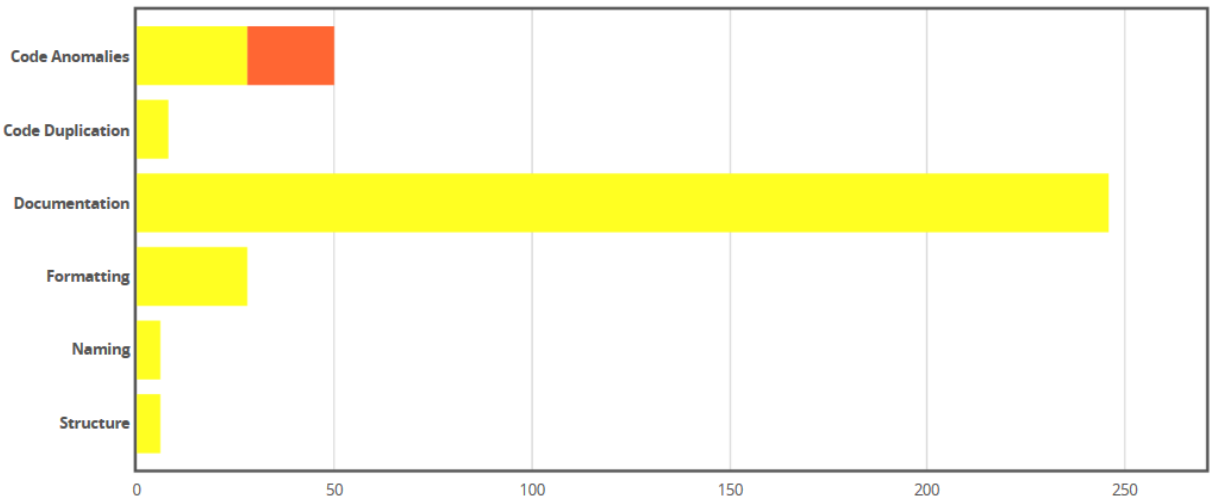

Abbildung 8.2: Von Teamscale gefundene Probleme von SIFCore nach dem Reengineering

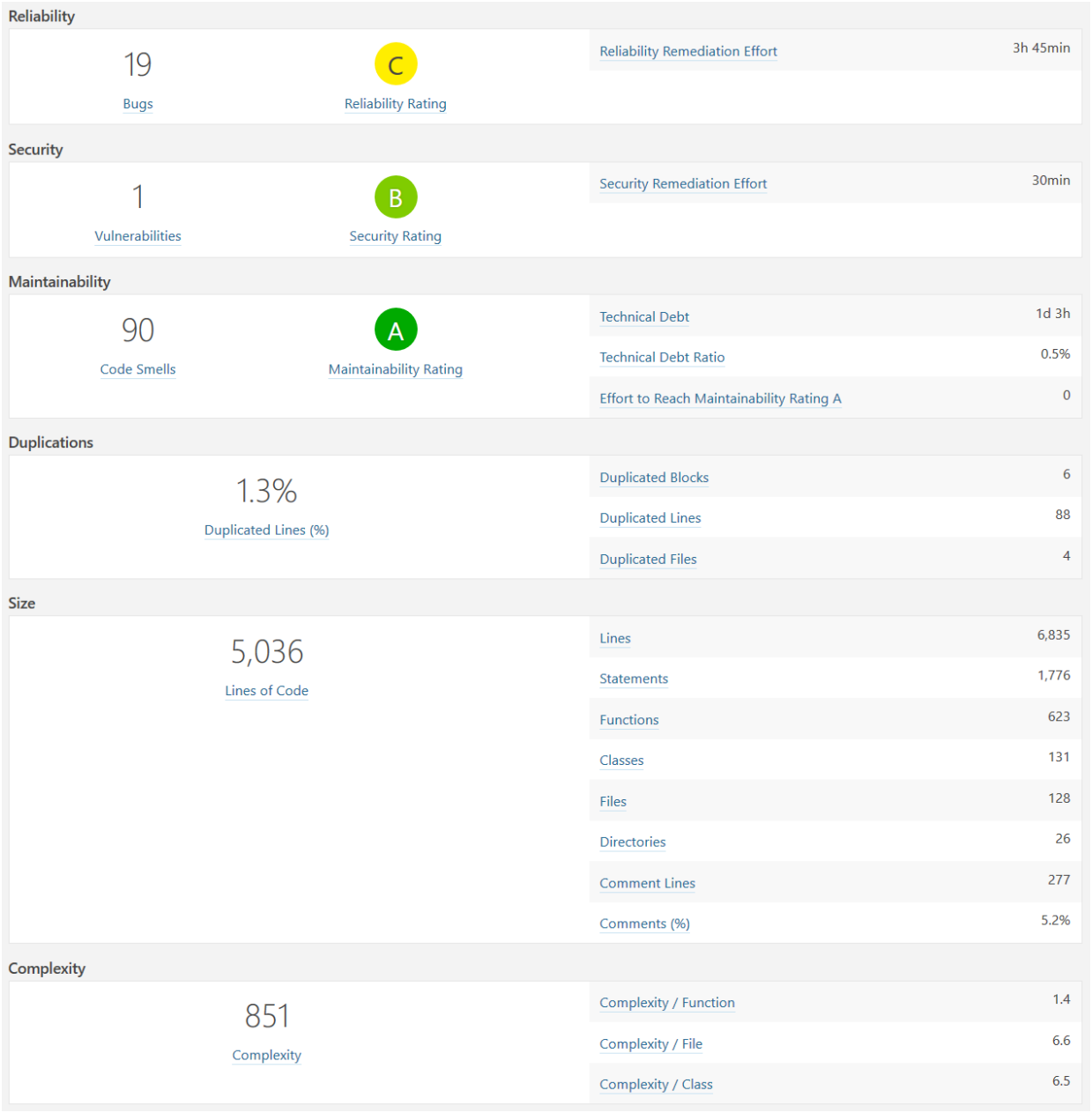

Abbildung 8.3: Bewertung von SIFCore durch Sonarqube nach dem Reengineering

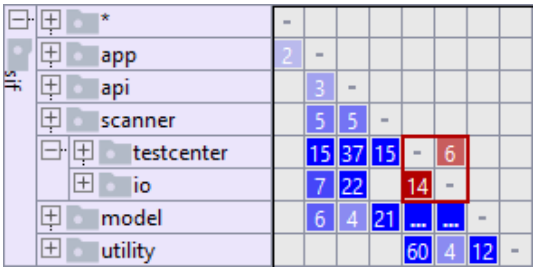

Abbildung 8.4: Dependency Structure Matrix von SIFCore nach dem Reengineering

Metriken haben erhebliche Defizite der Struktur aufgedeckt. Der Vergleich der Ergebnisse der beiden betrachteten Szenarien zeigt, dass mit der neuen Architektur deutlich weniger Stellen in der Software geändert werden müssen um die Szenarien erfüllen zu können.

Für das erste Szenario, die Erstellung einer neuen Prüfung, muss immer noch mindestens eine Komponente für die Ausführung der Prüfung (*Facility*), eine Komponente für die Konfiguration (*Policy*) und eine Komponente für gemeldete Verstöße (*Violation*) erstellt werden. Diese befinden sich nun jedoch, nach Prüfungen geordnet, in einem eigenen Modul. Auch existiert nur noch eine einzige Basis-*Facility*-Klasse und die umständliche Auslagerung der Bezeichner für das Austauschformat entfällt. Als wichtigste Verbesserung kann nun außerdem das Spreadsheet-Modell über eine einzige Schnittstelle, erstellt, aktualisiert und ausgeführt werden und es können nun alle Prüfungen statische und dynamische Aspekte des Spreadsheets prüfen.

Das zweite Szenario, das Unterstützen eines zusätzlichen Spreadsheet-Formats, konnte mit der alten Architektur nur mit großen Anpassungen durchgeführt werden. Grund dafür waren die viele verschiedenen Schnittstellen, über die auf das Spreadsheet zugegriffen wurde, und die teilweise auch von der eingesetzten POI-Bibliothek abhängig waren. Bei der neuen Architektur existiert nur noch eine Schnittstelle, die für das neue Spreadsheet-Format implementiert werden muss. Auch gibt es nun für einzelne Spreadsheet-Formate einen klar definierten Geltungsbereich, der unabhängig von anderen Formaten ist, da jeweils eigene Endpunkte des Webservice genutzt werden.

Die Auswertung der Einzelmetriken ergibt hinsichtlich der Strukturierung der Architektur ein ähnliches Bild. So sind die relativen und absoluten Werte der Metriken für die neue Architektur (Tabelle 8.1), verglichen mit der alten Architektur (Tabelle 4.1), durchgehend besser. Zu den wichtigsten Anhaltspunkten gehört hier die Metrik „Module mit wenigen Komponenten“. Während der absolute Wert deutlich kleiner ist als vor der Überarbeitung, konnte der relative Wert dieser Metrik nur leicht gesenkt werden. Allerdings wird in der neuen Struktur für jede Prüfung ein eigenes Modul angelegt, die mit drei Komponenten immer unter der Schranke dieser Metrik liegt und den Wert der Metrik bei insgesamt nur 26 Modulen so stark verzerrt. Wenn man diese elf Prüfungen nun herausrechnet, ergibt sich ein wesentlich besserer Wert für diese Metrik. Auch die „Maximale Verschachtelungstiefe von Modulen“ konnte deutlich von sieben auf zwei Ebenen gesenkt werden. Ebenso werden nach der Restrukturierung auch alle verfügbaren Klassen verwendet und nur vereinzelte Methoden werden nicht genutzt.

Auf Grund dieser Ergebnisse kann also angenommen werden, dass die Überarbeitung der Struktur erfolgreich war und die Architektur vereinfacht werden konnte, weil die Struktur der Module nun kompakter und flacher ist.

Überarbeitung der Kommunikations-Komponente

Der Umbau von SIFCore zu einer SOA mit Hilfe von REST hat zur Folge, dass es nun klar definierte Endpunkte gibt, mit denen die gewünschte Funktionalität einfach umgesetzt werden kann. Neue Endpunkte (wie etwa für neue Spreadsheet-Formate) können unabhängig erstellt werden, ohne dass Auswirkungen auf andere Module befürchtet werden müssen. Gleichzeitig ist SIFCore nun wieder vom eingesetzten Betriebssystem unabhängig und kann darüber hinaus auch wesentlich einfacher in eine Firmenumgebung integriert werden, weil für die Kommunikation auf etablierte und weit verbreitete Webtechnologien zurückgegriffen wurde.

Mit der Umsetzung als Servlet kann SIFCore nun auch Teil einer Cloud-Umgebung sein, da diese Architektur im Gegensatz zur alten Socket-Kommunikation auch mehrere unabhängige Clients gleichzeitig akzeptieren kann. Außerdem reduzieren die beiden neuen Austauschformate für Anfrage und Antwort, die das alte universelle Austauschformat ersetzen, den Overhead der Kommunikation weiter und sind auch besser strukturiert und somit leichter zu verstehen.

Entfernung der zentralen Metapher

Die Entfernung der zentralen Metapher war eine Maßnahme die getroffen wurde, weil vermutet wurde, dass die schlechte Strukturierung der Architektur eine Folge der zu komplizierten Metapher ist. Aus diesem Grund kann die Bewertung dieser Maßnahme auch nur indirekt erfolgen, weil der Negativbeweis, dass die Nichtnutzung einer zentralen Metapher sinnvoll ist, nicht erbracht werden kann. Der Umstand, dass die Restrukturierung der Module eine Komplexitätsreduzierung und Vereinfachung der Architektur ermöglichte, ohne dass diese dabei an eine bestimmte Metapher gebunden ist zeigt aber, dass auch ohne Metapher eine sinnvolle Strukturierung der Architektur möglich ist. Deshalb wird diese Maßnahme als Erfolg gewertet.

Verletzungen der Schichtenarchitektur beheben

Die ehemaligen Verletzungen der Schichtenarchitektur sind am deutlichsten bei der Auswertung der DSM (vergl. Abbildung 4.6 und Abbildung 8.4) zu sehen. Wenn man die beiden Matrizen vergleicht, wird sofort ersichtlich, dass nach der Überarbeitung von SIFCore die Matrix klarer und kompakter dargestellt werden kann. So hat die neue Matrix, abgesehen von den sechs eingetragenen zyklischen Abhängigkeiten zwischen den Modulen *IO* und *Testcenter*, die Form einer unteren Dreiecksmatrix. Das bedeutet, dass die Abhängigkeiten immer nur von oben nach unten bestehen und die Hierarchie der Module soweit eindeutig ist. Die verbliebenen zyklischen Abhängigkeiten sind der Implementierung geschuldet und können auch nicht aufgelöst werden: Der JAXB-Parser muss für einen erfolgreichen Import der *Policies* alle möglichen Spezialisierungen der Basisklasse

Policy kennen und verursacht so eine Abhängigkeit zwischen den einzelnen Prüfungen und dem *IO*-Modul.

Die Anzahl der Abhängigkeiten zwischen den Modulen konnte auch insgesamt gesenkt werden, es wurde also die Kopplung zwischen den Modulen verringert. Dementsprechend sind alle Verletzungen der Schichtenarchitektur, soweit es möglich war und die im Rahmen der Durchsicht und der Analyse der DSM gefunden wurden, aufgelöst und die Maßnahme wurde erfolgreich durchgeführt.

Überarbeitung der Spreadsheet-Schnittstellen

Die Überarbeitung der Spreadsheet-Schnittstellen hatte das Ziel einen einzigen Schnittpunkt zwischen dem *IO*-Modul und dem *Model*-Modul zu bilden, damit für die Bearbeitung und die Ausführung des Spreadsheets nicht mehr verschiedene Schnittstellen genutzt werden müssen. Zusätzlich sollte diese Schnittstelle unabhängig von der eingesetzten Bibliothek werden, damit die Unterstützung für weitere Spreadsheet-Formate ohne Änderung implementiert werden kann. Mit der Überarbeitung der Architektur konnte dieses Ziel erreicht werden, wie der Vergleich der Ergebnisse der szenariobasierten Analyseverfahren zeigt. Die Auflösung der Abhängigkeiten zwischen dem *IO*-Modul und dem *Model*-Modul kann außerdem beim Vergleich der DSM der beiden Architekturen beobachtet werden.

Überarbeitung der Fehlerbehandlung

Als letzte Maßnahme wurde die Überarbeitung der Fehlerbehandlung bestimmt, damit Fehler (insbesondere fehlerhafte Spreadsheets und Konfigurationsdaten) früh entdeckt und abgefangen werden, so dass der Zustand des internen Modells nicht kompromittiert werden kann. Weil die Grenze zwischen Architektur und Implementierung bei der Fehlerbehandlung fließend ist, kann der Erfolg dieser Maßnahme nicht direkt aus den (Architektur-) Analyseverfahren abgeleitet werden. Wenn die Fehlerbehandlung beispielsweise auf der Architekturebene korrekt geplant wurde, aber bei der Umsetzung ungenau oder fehlerhaft gearbeitet wird, kann diese stark beeinträchtigt werden. Dieses Problem zeigt sich aber bei der Architekturanalyse möglicherweise nicht, weil die Fehler nur auf der Implementierungsebene gemacht wurden. Aus diesem Grund werde ich mich bei der Bewertung dieser Maßnahme auch auf die Ergebnisse der Durchsicht des Quellcodes nach der Implementierung stützen, die zeigen, dass die durchgeführten Maßnahmen soweit auch korrekt umgesetzt wurden. Es gibt jedoch einige Metriken die Rückschlüsse auf die Qualität der Maßnahme erlauben: Die Werte für „Stille Exceptions“, „Verworfene Rückgabewerte von Methoden“ und „Anweisungen mit konstanten Bedingungen“ konnten alle verringert werden und dies kann als Verbesserung der Fehlerbehandlung interpretiert werden.

Gesamtbewertung

Nachdem die einzelnen Maßnahmen nun mit Hilfe der Analyseverfahren bewertet wurden, wird nun noch die Softwarequalität der gesamten Architektur untersucht. Dabei werden die

Ergebnisse der Pseudometriken verglichen, die mit Hilfe der der beiden Werkzeuge Sonarqube und Teamscale erhoben wurden.

Die Werte der von Teamscale erhobenen Metriken und Pseudometriken zeigen, dass der Umfang der Software verringert wurde. Außerdem werden für die neue Architektur erheblich weniger Probleme oder Fehler gemeldet, das Verhältnis von Fehlern pro Codezeile ist jedoch ungefähr gleich geblieben. Bei dieser Betrachtung wird jedoch nicht die Schwere der Fehler berücksichtigt, sondern nur die Anzahl. Bei der Aufschlüsselung der einzelnen Probleme nach Kategorie und Fehlergrad wird jedoch deutlich, dass die Anzahl der „kritischen“[1] Fehler (vergl. Abbildung 4.4 und Abbildung 8.2) absolut und relativ gesehen bei der neuen Architektur kleiner ist. Auch die Fehler in den einzelnen Kategorien sind, mit Ausnahme der Kategorie „Documentation“, absolut und relativ gesehen weniger geworden. Die Anzahl der Probleme in der Kategorie „Documentation“ ist hingegen nur leicht zurückgegangen. Bei dem deutlich geringerem Umfang von SIFCore in der neuen Version erhöht sich der relative Anteil dieser Probleme enorm. Der Grund für das schlechte Abschneiden von SIFCore liegt darin, dass auf eine formale Javadoc-Dokumentation weitgehend verzichtet wurde und stattdessen an wichtigen Punkten eine weniger formale, aber dafür ausführlichere Dokumentation geschrieben wurde. Für eine gute Bewertung durch Teamscale in dieser Kategorie, ist aber eine solche formale Dokumentation notwendig.

Als nächstes werden die als Kuchendiagramme (vergl. Abbildung 4.3 und Abbildung 8.1) dargestellten Pseudometriken zur Methodenlänge und Verschachtelungstiefe von Methoden untersucht. In der neuen Architektur existieren für beide Metriken keine *schwerwiegenden* Probleme mehr und auch die Anteile der Methoden die Warnungen erzeugen ist deutlich zurück gegangen. Die Bewertung von SIFCore durch Teamscale ist nach der Überarbeitung also insgesamt besser, diese Ergebnisse decken sich mit den Ergebnissen der Bewertung der Einzelmaßnahmen.

Als letztes werden jetzt noch die Metriken und Pseudometriken von Sonarqube (Abbildung 8.3 und Abbildung 4.5) verglichen. Der Wert der Pseudometrik für die Zuverlässigkeit hat sich dabei am deutlichsten geändert: Von der schlechtesten Note „E“ auf ein mittleres „C“. Gleichzeitig ist die Zahl der gefundenen Bugs von 169 auf 19 gesunken. Der Umfang der Software wird dabei von Sonarqube ähnlich wie von Teamscale bewertet, auch wenn für die Anzahl der Codezeilen hier strengere Kriterien verwendet werden. Auch hier ist der gemessene Umfang von SIFCore um ca. 50% zurückgegangen, das bedeutet eine Verbesserung der relativen und absoluten Anzahl an gefundenen Bugs.

Die Bewertung der beiden anderen Qualitätsmerkmale hat sich hingegen nicht geändert, weil diese (trotz der hohen Fehleranzahl) bereits „B“- und „A“-Noten erhalten haben. Trotzdem wurden auch hier ausnahmslos die Anzahl der gefundenen Probleme verringert.

Die erhoffte Vereinfachung der Architektur und Reduzierung der Komplexität durch die Überarbeitung von SIFCore wird auch durch die Untersuchung der von Sonarqube erhobenen Pseudometriken gestützt. Die Werte weisen bei der Gesamtkomplexität eine Verringerung um 44% auf und auch die Werte für die Durchschnittskomplexität von Methoden oder Klassen sind gesunken.

[1] mit roter Farbe gekennzeichnet

Die Ergebnisse der Metriken und Pseudometriken sind meiner Meinung nach eindeutig und zeigen eine Verbesserung der Softwarequalität von SIFCore durch die Überarbeitung. Auch decken sich die Ergebnisse der beiden Werkzeuge bei der Gesamtbewertung, sodass eine höhere Wahrscheinlichkeit besteht, dass diese Ergebnisse zuverlässig sind. Die Bewertung der Einzelmaßnahmen kommt dabei zum gleichen Schluss und ich interpretiere die Ergebnisse der Analyseverfahren dahingehend, dass die gesetzten Ziele erreicht und die Softwarequalität gesteigert werden konnte.

9 Evaluation der Analyseverfahren

In diesem Kapitel wird der Einsatz der verwendeten Analyseverfahren kritisch hinterfragt. Es wird eine Einschätzung der Qualität der Ergebnisse dieser Verfahren gegeben und es wird versucht, die einzelnen Verfahren gegeneinander abzugrenzen. Für alle Verfahren werden dann die gefundenen Vor- und Nachteile diskutiert. Es sollen dabei auch die in Abschnitt 1.2.1 formulierten Fragen beantwortet werden.

9.1 Szenarien

Im Rahmen dieser Arbeit wurden die zwei szenariobasierten Analyseverfahren SAAM und ALMA angewendet. Bei der Evaluation hat sich herausgestellt, dass sich die Ergebnisse der beiden Verfahren so sehr ähneln, dass es wenig sinnvoll erscheint, diese voneinander zu trennen. Ein Grund für diese Überschneidung ist, dass für beide Verfahren die gleichen Szenarien verwendet wurden und diese auch nach den gleichen Merkmalen evaluiert wurden. Dies war gewünscht und beabsichtigt, damit eventuelle Unterschiede der Verfahren bewertet werden können. Es zeigt aber, dass die Unterschiede der beiden Verfahren nicht bei der eigentlichen Auswertung der Szenarien liegen, sondern hauptsächlich bei der Auswahl der Szenarien und der allgemeinen Vorgehensweise. Ein Unterschied, der zwischen beiden Verfahren gefunden wurde, ist dass bei ALMA die Formulierung der Ziele im Vordergrund steht und die Erfüllung dieser ein zentrales Merkmal ist, während die Zielsetzung bei SAAM nur indirekt über eine Priorisierung der Szenarien zum Ausdruck kommt.

Die bereits in Kapitel 4 vorgestellten Studien ([Kaz+96] und [Las02]), die untersuchen wie SAAM und ALMA sich bei real existierenden Softwareprojekten eingesetzt werden, haben dabei auch mit dem Problem zu kämpfen, dass die Verfahren von den Projektbeteiligten unterschiedlich interpretiert wurden. Aus diesen Informationen und der Erfahrung, die bei der Durchführung der beiden Verfahren gewonnen wurde, schließe ich, dass die Unterschiede der Verfahren nicht so wichtig für die Ergebnisse sind wie eine gründliche Vorbereitung (in der der Analyseprozess genau festgelegt wird) und eine sorgfältige Durchführung der Szenario-Evaluation.

Leider verzichten beide Verfahren auf eine ausführliche Beschreibung des Analyseprozesses. Bei der Wahl der geeigneten Werkzeuge und Beschreibungssprachen, die durchaus relevant für die Durchführung sind, wird man bei beiden Verfahren auch nur sehr oberflächlich unterstützt. Diese und weitere Details des Analyseprozesses müssen deshalb den jeweiligen Softwareprojekten selbst bestimmt werden, in denen die Analyse durchgeführt wird. Aus diesem Grund gibt es keinen Standardprozess für diese Analyseverfahren und es ist in der Praxis kaum mehr möglich zwischen SAAM und ALMA zu unterscheiden.

SAAM und ALMA eigenen sich laut Aussage ihrer Autoren beide für eine frühe Analyse der Architektur, also bevor diese implementiert wird. Wenn keine Implementierung vorhanden ist, kann sich die Analyse nur auf die vorhandene Architekturdokumentation stützen. Beide Methoden gehen bei der Analyse aber durchaus auf Details der Architektur ein und fordern zugleich eine recht formale Beschreibung der Architektur. Das müssen sie auch, da sonst eine sinnvolle Evaluation der Szenarien nicht möglich ist und die Aussagekraft ihrer Ergebnisse leidet. Meiner Erfahrung nach wird in der Praxis jedoch meist auf eine ausführliche und formal korrekte Dokumentation der Architektur verzichtet. Dies geschieht meist zugunsten der Implementierung, besonders wenn der Zeit- oder Kostendruck hoch ist.

Deshalb erfordert ein auf Szenarien basierendes Analyseverfahren meist eine hohe Einarbeitungszeit, in der ein Großteil der Zeit für die Beschreibung der Architektur verwendet werden muss. Auch besteht die Gefahr, dass man, anstatt die Architektur nur zu beschreiben, eine Architektur entwickelt, also aktiv Details oder Merkmale hinzufügt die davor nicht existierten. Dafür bieten diese Verfahren den Vorteil, dass sie feine und spezifische Unterscheidungen zwischen verschiedenen möglichen Lösungen für Probleme der Architektur sichtbar machen können (vergl. Abschnitt 1.2.1 Punkt 2), weil die Szenarien auch auf einzelne Details der Architektur eingehen.

Auch bei SIFCore bestand das Problem, dass für eine genaue Analyse mit Szenarien die bisherige Beschreibung der Architektur nicht ausreichend und detailliert genug war. Es konnte die Architekturdokumentation auf Basis der vorhandenen Implementierung aber in der gewünschten Detailtiefe wiederhergestellt werden. Wenn diese Analyseverfahren bei einem Reengineering (vergl. Abschnitt 1.2.1 Punkt 1) eingesetzt werden, kann der hohe Aufwand für die Beschreibung der Architektur immerhin einige Synergieeffekte für andere Aktivitäten des Entwicklungsprozesses zur Folge haben. Im Falle von SIFCore wurden die Ergebnisse der Architekturrekonstruktion außer für SAAM und ALMA auch für die Formulierung der Anforderungen verwendet. Dabei konnte das Verständnis von SIFCore, durch die intensive Vorbereitung auf die Analyseverfahren, gesteigert werden, was den Entwurf der neuen Architektur vereinfachte. Insofern konnte der Aufwand gerechtfertigt werden und wird von mir auch als sinnvoll angesehen. Zumal irgendeine Aktivität für das Programmverstehen durchgeführt werden muss, um die Software sinnvoll verändern zu können, auch wenn auf szenariobasiertes Analyseverfahren verzichtet wird.

Schlussendlich hat die Evaluation der Szenarien Erkenntnisse über SIFCore geliefert, die mit auf Metriken basierenden Verfahren nicht möglich gewesen wären. So wurden die Rollen der einzelnen Spreadsheet-Schnittstellen bei der Prüfung aufgeschlüsselt und diese konnten dadurch überarbeitet und zusammengefasst werden. Auch bei der Restrukturierung der Module haben die Erkenntnisse von SAAM und ALMA eine wichtige Rolle gespielt.

9.2 Metriken

Es wurden im Rahmen des Reengineerings ausgewählte Einzelmetriken, Pseudometriken und die DSM genutzt um die Architektur von SIFCore zu bewerten. Alle diese Verfahren untersuchen dabei die Architektur als Ganzes, analysieren also allen vorhandenen Quellcode (und teilweise den kompilierten Binärcode) und zählen oder schätzen, meist mit Hilfe von Werkzeugen, bestimmte

Merkmale, die dann als numerische Werte zur Verfügung gestellt werden. Durch dieses Vorgehen ergeben sich verschiedene Vor- und Nachteile von auf Metriken basierenden Verfahren.

Diese auf Metriken basierende Verfahren können mit geringem Aufwand und wenig Vorbereitung durchgeführt werden. Sofern der Quellcode vorhanden ist und die Werkzeuge funktionieren, können also schnell Ergebnisse erzielt werden. Dabei müssen aber die in Abschnitt 4.3 genannten, häufigsten Fehler bei der Verwendung von Metriken beachtet werden, was dazu führt, dass entweder der Entwickler über genügend Vorwissen verfügen muss oder aber die Vorbereitungszeit wieder ansteigt.

Auch bei SIFCore mussten zuerst die geeigneten Einzelmetriken ausgesucht werden, mit denen die Qualitätsmerkmale *Änderbarkeit* und *Zuverlässigkeit* bestimmt werden können. Dabei war die größte Schwierigkeit, Metriken zu finden, die den aktuellen Stand der Architektur repräsentieren und dabei gleichzeitig auch Schwächen aufdecken, die dann gezielt verbessert werden können. Diese Auswahl ist jedoch ohne eine gute Kenntnis der Architektur dem Zufall überlassen, und so steigt auch bei diesen Verfahren die Nützlichkeit mit dem Aufwand der in die Vorbereitung gesteckt wird.

Bei der Analyse der SIFCore-Metriken konnte außerdem beobachtet werden, dass sich die Werte von verschiedenen Werkzeugen teilweise deutlich unterscheiden. So wurde zum Beispiel der Umfang der Software bei allen Werkzeugen mit der Metrik *LoC* bestimmt. Für diese Metrik wurden von Teamscale (für die alte Version von SIFCore) 18.000 Zeilen gezählt, während Sonarqube als Wert nur 10.000 Zeilen angegeben hat. Dies zeigt, dass bei der Verwendung von Metriken immer auf die genaue Definition geachtet werden muss und nur Metriken, die nach der gleichen Definition erhoben wurden, auch verglichen werden können. Besonders davon betroffen sind Pseudometriken, weil diese nach einem komplexeren Modell berechnet werden. Der so berechnete Wert ist natürlich davon abhängig, welche Annahmen und Voraussetzungen in diesem Modell verwendet werden und diese unterscheiden sich je nach Werkzeug.

Als letztes Merkmal von Metriken ist bei der Untersuchung von SIFCore aufgefallen, dass es sehr schwer ist den Erfolg von einzelnen Maßnahmen mit Hilfe von Metriken zu messen, weil die Metriken in der Regel immer die gesamte Software erfassen. Zwar können auch Metriken für einzelne Module oder Klassen erhoben werden, aber damit steigt dann wiederum der Aufwand und die Anzahl der zu überwachenden Metriken, was man aber vermeiden möchte.

Wenn beim Einsatz von Metriken alle möglichen Probleme beachtet und entsprechende Gegenmaßnahmen ergriffen werden, dann empfehle ich eine Verwendung dieser Analysemethoden im Rahmen eines Reengineerings (vergl. Abschnitt 1.2.1 Punkt 1), da diese eine zeitsparende Methode zur allgemeinen Bewertung der Architektur ermöglichen. Eine rein auf Metriken basierende Architekturanalyse kann ich aber nicht empfehlen, da diese nicht alle Aspekte einer Architektur untersuchen kann und insbesondere bei Sonderfällen versagt (vergl. Abschnitt 1.2.1 Punkt 2). Dabei sind gerade diese Sonderfälle oft der Grund für Probleme mit der Architektur.

Als besonders hilfreiche, einzelne Metrik habe ich die DSM empfunden, weil diese eine kompakte und dennoch verständliche Darstellung der Architektur und ihrer Schichten erlaubt. Gegenüber einem Graphen hat die DSM den Vorteil, quantitative Aussagen einfacher darstellen zu können und gleichzeitig einen Überblick über die gesamte Architektur zu bieten.

10 Fazit

In diesem Kapitel werden nach einem kurzem Rückblick auf den Verlauf der Arbeit die erreichten Ergebnisse zusammengefasst und mit den in Kapitel 1 formulierten Zielen verglichen. Zum Schluss gibt es einen Ausblick auf mögliche zukünftige Arbeiten am Spreadsheet Inspection Framework.

10.1 Rückblick

In dieser Arbeit wurde die Software SIFCore einer umfassenden Analyse und anschließender Überarbeitung unterzogen. Auf der Basis der bisherigen Entwicklungsgeschichte von SIFCore, der Aufgabenstellung der Arbeit, einem Interview mit dem Projektleiter und einem Livetest der Software wurden die Meilensteine der Arbeit und die Ziele für die Überarbeitung festgelegt. Diese Ziele umfassten dabei die Verbesserung der Softwarequalität von SIFCore, insbesondere der Qualitätsmerkmale Änderbarkeit und Zuverlässigkeit. Neben der Bewertung der Software sollten aber auch die verwendeten Verfahren selbst untersucht werden, damit ein Vergleich und eine qualitative Aussage über die Verfahren möglich ist (vergl. Kapitel 1).

Zu Beginn der Arbeit wurde untersucht, welche Analyseverfahren im Rahmen eines Reengineerings eingesetzt und welche Ziele damit erreicht werden können. Dabei wurde festgestellt, dass die Anzahl der verfügbaren Analyseverfahren sehr hoch ist und es keinen Konsens in der Forschung gibt, welche Verfahren effizient sind und zu guten Ergebnissen führen. Aus diesem Grund wurden die verfügbaren Verfahren in Kategorien eingeteilt und dann einige, als geeignet erscheinende Verfahren ausgewählt (vergl. Kapitel 4).

Zu diesem Zeitpunkt wurde ebenfalls eine erste Sichtung der vorhandenen Architekturdokumentation und des vorhandenen Quellcodes durchgeführt. Dabei wurde festgestellt, dass die Beschreibung der Architektur nicht vollständig und detailliert genug für eine Analyse ist. Ich habe mich deshalb entschieden, vor der eigentlichen Analyse zuerst eine Architekturrekonstruktion durchzuführen, damit die einzelnen Beschreibungen der Architektur, die aus vielen verschiedenen Abschlussarbeiten stammen, zusammengefasst und auf die gewünschte Detailtiefe gebracht werden können (vergl. Kapitel 3).

Nach der Architekturrekonstruktion, wurden die einzelnen Analyseverfahren durchgeführt. Mit SAAM und ALMA wurden zwei Verfahren verwendet, die untersuchen welche Änderungen an einer Software für die Durchführung eines Szenarios benötigt werden und welche Auswirkungen diese haben. Diese Ergebnisse, zusammen mit den mittels Werkzeugen erhobenen Metriken und

der DSM, wurden dann genutzt, um die Anforderungen an die überarbeitete Version von SIFCore zu formulieren (vergl. Kapitel 5).

Es wurden verschiedene Architekturkonzepte und Technologien untersucht, mit denen diese Anforderungen möglichst einfach umgesetzt werden können. Zu diesem Zeitpunkt wurde festgelegt, dass eine SOA mit Hilfe von REST umgesetzt werden soll. Es wurden dazu verschiedene, verfügbare Bibliotheken getestet, bevor auf die Referenzimplementierung *Jersey* zurückgegriffen wurde. Ebenfalls wurden verschiedene DI-Frameworks evaluiert, um eine Lösung zu finden, die auch langfristig alle Anforderungen erfüllen kann. So fiel die Wahl letztendlich auf das *Google Guice*-Framework.

Nachdem die Grundkomponenten soweit entworfen waren wurde mit der Implementierung der Architektur begonnen, die zu diesem Zeitpunkt das DI-Framework und den Webserver beinhaltete. In vier aufeinanderfolgenden Iterationen wurden dann die weiteren Bestandteile von SIFCore entworfen und anschließend implementiert (vergl. Kapitel 6 und Kapitel 7): Zuerst die IO-Komponente für Spreadsheets, dann das Austauschformat, dann die Prüfungen, und zum Schluss das überarbeitete Spreadsheet-Modell. Während dieser Phasen wurden auch nach und nach die Modul- und Integrationstests implementiert, damit die Funktionalität der einzelnen Komponenten so früh wie möglich sichergestellt werden kann.

Aufgrund der sehr umfangreichen Entwurfsphasen für die Überarbeitung der Komponenten habe ich mich entschieden, die beiden ursprünglich getrennten Entwurfs- und Implementierungsphasen für die Überarbeitung der Architektur und die funktionale Verbesserung von SIFCore in einem Schritt zusammenzufassen. Es wurde also gleichzeitig die Architektur überarbeitet und es wurden einige funktionale Verbesserungen hinzugefügt.

Nachdem die Überarbeitung von SIFCore abgeschlossen war, wurden die Architekturanalyseverfahren erneut durchgeführt (vergl. Kapitel 8). Die Ergebnisse der beiden Analysen wurden dann miteinander verglichen, um festzustellen ob und in welchem Umfang sich die Qualität der Software verbessert hat. Dabei haben die Ergebnisse aller Analyseverfahren bestätigt, dass die Maßnahmen erfolgreich umgesetzt und die Softwarequalität verbessert werden konnten.

In einem letzten Meilenstein wurden dann die Analyseverfahren selbst noch untersucht und bewertet. Dabei wurde darauf geachtet, dass die Ergebnisse der einzelnen Verfahren gegeneinander abgegrenzt werden, aber auch die Prozesse der Verfahren untersucht werden. Es wurde festgestellt, dass die Verfahren zum Teil unterschiedliche Anforderungen an die zu untersuchende Software haben und sie sich auch in Aufwand und den untersuchten Merkmalen deutlich unterscheiden (vergl. Kapitel 9).

10.2 Bewertung

Die in Abschnitt 1.2 formulierten Ziele dieser Arbeit wurden soweit alle erreicht. Die Architektur von SIFCore konnte durch die Verwendung von unterschiedlichen Analyseverfahren untersucht und bewertet werden. Dabei wurden sowohl szenariobasierte Analyseverfahren, als

auch auf Metriken basierende Verfahren eingesetzt, die jeweils unterschiedliche Schwerpunkte der Architektur evaluierten.

Das Reengineering von SIFCore, also die Identifizierung der Schwachstellen und die anschließende Überarbeitung von SIFCore führten zu einer Verbesserung der Softwarequalität. Die wurde hauptsächlich dadurch erreicht, dass die Komplexität der Architektur verringert und auch der Umfang des Quellcodes deutlich reduziert werden konnte, was in Kapitel 8 bei der Bewertung der Architektur festgestellt wurde. Die Auswertung der Analyseverfahren SAAM und ALMA hat ergeben, dass mit der neuen Architektur die Szenarien einfacher umgesetzt werden können. Die Ergebnisse der Metriken zeigen, dass eine Verbesserung der Architektur in vielen Punkten möglich war. Die vollständige Funktionalität von SIFCore gegenüber der Vorgängerversion wurde durch Modul- und Integrationstests sichergestellt.

Es wurden außerdem einige funktionale Verbesserungen an SIFCore vorgenommen. So können jetzt unter anderem einzelne Zellen oder ganze Arbeitsblätter bei der Ausführung von Prüfungen ignoriert werden. Alle Prüfungen funktionieren nun korrekt für Spreadsheets mit mehreren Arbeitsblättern und die Prüfungen bieten mehr Konfigurationsmöglichkeiten.

Gleichzeitig wurde viel Aufwand in die Dokumentation der neuen Architektur gesteckt, damit das Verständnis der Entwickler für die Software verbessert werden kann und so verhindert wird, dass mit der Weiterentwicklung von SIFCore wieder eine rapide Verschlechterung der Softwarequalität eintritt. Es wurde versucht die Einstiegshürden für neue Entwickler möglichst gering zu halten und die Komplexität der Architektur soweit wie möglich zu reduzieren. Zwar sind diese Vorkehrungen keine Garantie für einen langsameren oder gar gestoppten Verfall der Architektur in weiteren Entwicklungsphasen, ich bin jedoch zuversichtlich, dass die positiven Effekte dieses Reengineerings in zukünftigen Versionen von SIFCore sichtbar werden.

Neben den genannten praktischen Verbesserungen von SIFCore sollten auch noch die in Abschnitt 1.2.1 formulierten Fragen beantwortet werden. Diese Fragen konnten zwar in Kapitel 8 und Kapitel 9 teilweise beantwortet werden, aber ich stelle auch fest, dass es nicht möglich ist eine allgemeingültige Aussagen zu treffen, wie hilfreich diese Analyseverfahren für ein Reengineering sind. So mussten die Verfahren teilweise stark angepasst werden, weil diese nicht für ein Ein-Mann-Projekt ausgelegt sind und zum anderen ist die Umsetzung dieser Verfahren oft nicht detailliert festgelegt, so dass ein Vergleich mit anderen Softwareprojekten schwer fällt.

Die gleichzeitige Durchführung von mehreren Analyseverfahren und vor allem die ausführliche Vorbereitung in Form einer Architekturrekonstruktion haben dazu geführt, dass eine klare Abgrenzung zwischen den Verfahren nicht mehr möglich ist, weil Ergebnisse eines Verfahrens (teilweise unbewusst) auch für andere Verfahren genutzt werden.

Die Bewertung der Analyseverfahren kann dennoch einige, wenn auch bereits in anderen Arbeiten beschriebene, Erkenntnisse liefern. Die Vorbereitungs- und Einarbeitungszeit ist für szenariobasierte Verfahren höher als für andere Verfahren, dafür sind die Ergebnisse auch detaillierter. Außerdem kann die benötigte Zeit verringert werden, wenn die Architekturdokumentation vollständig und detailliert genug ist. Die investierte Zeit kann außerdem auch als gleichzeitige Vorbereitung für die Entwurfs- oder Implementierungsphase dienen. So hat sich im Falle von SIFCore die Architekturrekonstruktion gelohnt und es konnten wichtige Informationen aufbereitet werden, bevor diese dann an unterschiedlichen Stellen im Projekt verwendet wurden.

Die Verfahren die auf Metriken basieren können mit Hilfe von Werkzeugen schnell durchgeführt werden. Die Auswahl der einzelnen Metriken und die Interpretation der Werte benötigen jedoch auch Vorkenntnisse oder eine entsprechende Vorbereitung, womit die Zeitersparnis wieder relativiert wird oder in Kauf genommen wird, dass die Ergebnisse nicht aussagekräftig sind.

Abschließend kann ich empfehlen, dass für ein Reengineering immer eine Kombination von verschiedenen Analyseverfahren eingesetzt werden sollte, um die Vor- und Nachteile der einzelnen Verfahren ausgleichen zu können und so die Kosten für eine Verbesserung der Softwarequalität zu begrenzen.

10.3 Zukünftige Arbeiten

Arbeit an Software ist grundsätzlich nie abgeschlossen. Die Architektur von SIFCore ist jedoch soweit verbessert worden, dass ich mir aktuell nur noch kleinere Veränderungen vorstellen kann. So gibt es im Moment keine Authentifizierungs- und Zugriffskontrolle für die Nutzung der Prüfungen. Durch die nun hinzugekommene Unterstützung für mehrere gleichzeitige Benutzer und die Umsetzung von SIFCore als Webservice könnte eine solche Architekturerweiterung aber in Zukunft durchaus benötigt werden. Auch kann die technische Schuld von SIFCore mit entsprechendem Aufwand noch weiter reduziert werden, aber diese ist bereits auf einem sehr niedrigen Stand. Deshalb sehe ich bei SIFCore hauptsächlich den Bedarf für weitere funktionale Erweiterungen. Auf Basis der Erfahrung mit Metriken zur Analyse von Software schlage ich eine neue Komponente für SIFCore vor. Diese könnte Metriken von Spreadsheets erfassen, um mit diesen, zusätzlich zu den bisherigen Prüfungen, das Spreadsheet bewerten zu können. Die dafür benötigten Komponenten sind mit den *Scannern* bereits vorhanden und auch eine Erweiterung des XML-Austauschformats zum Transport der Metriken sollte entsprechend leicht fallen.

Abgesehen von den Verbesserungen an der Architektur von SIF könnten zukünftige Arbeiten tiefergehend untersuchen, ob der Einsatz von auf Szenarien basierenden Analyseverfahren für solche sehr kleinen Softwareprojekte auch dann noch sinnvoll ist, wenn kein Reengineering sondern eine Neuentwicklung durchgeführt wird. In einem solchen Fall könnte die Architekturdokumentation nicht wiederhergestellt werden, die Nebeneffekte der intensiven Vorbereitung könnten nicht so gut für andere Verfahren genutzt werden, und das Problem, dass die Verfahren für Einzelentwickler ohne Team stark angepasst werden müssen, bestünde nach wie vor.

Auch die Vorbereitung für szenariobasierte Analyseverfahren könnte noch genauer untersucht werden. So ist im Moment nicht eindeutig genug erforscht, welche Architekturbeschreibungen besonders nützlich sind, welche Sprachen und Werkzeuge sich für diese Beschreibung am besten eignen und auf welche Artefakte für diese Analysen verzichtet werden kann. Mit einem solchen Verständnis würde sich die benötigte Zeit für Vorbereitung und Ausführung reduzieren und die Analyse der Architektur beschleunigen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung	Erstes Vorkommen
ADL	Architecture Description Language	30
ALMA	Architecture Level Modifiability Analysis	39
DI	Dependency Injection	22
DSM	Dependency Structure Matrix	40
HTTP	Hypertext Transfer Protocol	22
HTTPS	Hypertext Transfer Protocol Secure	22
JVM	Java Virtual Machine	64
LoC	Lines of Code	50
M2M	Machine-to-Machine	65
ODF	OASIS Open Document Format for Office Applications	44
REST	Representational State Transfer	22
SAAM	Software Architecture Analysis Method	39
SIF	Spreadsheet Inspection Framework	11
SIFCore	Spreadsheet Inspection Framework Service	24
SIFEI	Spreadsheet Inspection Framework Excel AddIn	24
SOA	Service-oriented Architecture	21
SOAP	Simple Object Access Protocol	22
SRP	Single Responsibility Principle	22
UML	Unified Modeling Language	30

Literaturverzeichnis

[Abo+97] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski. *Recommended Best Industrial Practice for Software Architecture Evaluation*. Techn. Ber. CMU/SEI-96-TR-025. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12653 (zitiert auf S. 33).

[Ale+77] C. Alexander, S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, I. Fiksdahl-King. *A pattern language*. Gustavo Gili, 1977 (zitiert auf S. 18).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. 2. Aufl. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154959 (zitiert auf S. 17, 20).

[BG] M. Babar, I. Gorton. „Comparison of Scenario-Based Software Architecture Evaluation Methods“. In: *11th Asia-Pacific Software Engineering Conference*. Institute of Electrical und Electronics Engineers (IEEE). DOI: 10.1109/apsec.2004.38. URL: http://dx.doi.org/10.1109/APSEC.2004.38 (zitiert auf S. 39).

[BZJ04] M. A. Babar, L. Zhu, R. Jeffery. „A Framework for Classifying and Comparing Software Architecture Evaluation Methods“. In: *Proceedings of the 2004 Australian Software Engineering Conference*. ASWEC '04. Washington, DC, USA: IEEE Computer Society, 2004, S. 309–. ISBN: 0-7695-2089-8. URL: http://dl.acm.org/citation.cfm?id=987680.987740 (zitiert auf S. 21).

[Bec14] S. Beck. *Spreadsheet-Fehlermuster*. 2014. DOI: 10.18419/opus-3306. URL: https://doi.org/10.18419/opus-3306 (zitiert auf S. 22, 27).

[Ben+] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet. „Architecture-level Modifiability Analysis (ALMA)“. In: *J. Syst. Softw.* (). DOI: 10.1016/S0164-1212(03)00080-3. URL: http://dx.doi.org/10.1016/S0164-1212(03)00080-3 (zitiert auf S. 39).

[Bou13] E. Bouwers. „Metric-based Evaluation of Implemented Software Architectures“. In: (2013). DOI: 10.4233/uuid:6b65c5f5-398c-4a41-8806-31c638b1891c (zitiert auf S. 34).

[CB89] W. Cunningham, K. Beck. „Constructing abstractions for object-oriented applications.“ In: *Journal of Object-Oriented Programming* 2.2 (1989), S. 17–19 (zitiert auf S. 18).

[CI90] E. J. Chikofsky, J. H. C. II. „Reverse Engineering and Design Recovery: A Taxonomy“. In: *IEEE Softw.* 7.1 (1990), S. 13–17. ISSN: 0740-7459. DOI: http://dx.doi.org/10.1109/52.43044 (zitiert auf S. 14).

[DN02] L. Dobrica, E. Niemelä „A Survey on Software Architecture Analysis Methods“. In: *IEEE Trans. Softw. Eng.* 28.7 (2002), S. 638–653. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1019479. URL: http://dx.doi.org/10.1109/TSE.2002.1019479 (zitiert auf S. 21).

[Deu02] A. van Deursen. „Software Architecture Recovery and Modelling: [WCRE 2001 Discussion Forum Report]“. In: *SIGAPP Appl. Comput. Rev.* 10.1 (2002), S. 4–7. ISSN: 1559-6915. DOI: 10.1145/568235.568236. URL: http://doi.acm.org/10.1145/568235.568236 (zitiert auf S. 20).

[Dou13] E. Doust. *Visualisierung von Fehlern in Spreadsheets*. 2013. DOI: 10.18419/opus-3278. URL: https://doi.org/10.18419/opus-3278 (zitiert auf S. 22, 27, 53, 55).

[Dud] *Duden – Deutsches Universalwörterbuch*. 6. Aufl. Mannheim: Bibliographisches Institut, Okt. 2006. ISBN: 3411055065. URL: http://www.duden.de/suche/detail.php?isbn=3-411-05506-5 (zitiert auf S. 15).

[Erl+12] T. Erl, B. Carlyle, C. Pautasso, R. Balasubramanian. *SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST*. 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2012. ISBN: 0137012519, 9780137012510 (zitiert auf S. 19).

[FB99] M. Fowler, K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999 (zitiert auf S. 41).

[FT02] R. T. Fielding, R. N. Taylor. „Principled Design of the Modern Web Architecture“. In: *ACM Trans. Internet Technol.* 2.2 (2002), S. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185 (zitiert auf S. 19).

[Fow04] M. Fowler. „Inversion of control containers and the dependency injection pattern“. In: (2004) (zitiert auf S. 19).

[Gam+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (zitiert auf S. 18).

[Gar84] D. A. Garvin. „What Does ‚Product Quality‘ Really Mean?“ In: (1984) (zitiert auf S. 15).

[Ieea] *IEEE Recommended Practice for Software Requirements Specifications*. DOI: 10.1109/ieeestd.1998.88286. URL: https://doi.org/10.1109%2Fieeestd.1998.88286 (zitiert auf S. 17, 20, 23).

[Ieeb] *International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering*. DOI: 10.1109/ieeestd.2006.235774. URL: http://dx.doi.org/10.1109/IEEESTD.2006.235774 (zitiert auf S. 13).

[Iso] *Qualitätsmanagement, Qualitätsmanagementsysteme Grundlagen und Begriffe (ISO 9000:2005)* (zitiert auf S. 15).

[Kaz+94] R. Kazman, L. Bass, M. Webb, G. Abowd. „SAAM: A Method for Analyzing the Properties of Software Architectures“. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. 1994, S. 81–90. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257746 (zitiert auf S. 37).

[Kaz+96] R. Kazman, G. Abowd, L. Bass, P. Clements. „Scenario-based analysis of software architecture“. In: *IEEE software* 13.6 (1996), S. 47–55 (zitiert auf S. 36–38, 89).

[Kra14] W. Kraus. *Plausibilitätsprüfung implizit gekoppelter Spreadsheet-Daten*. 2014. DOI: 10.18419/opus-3357. URL: https://doi.org/10.18419/opus-3357 (zitiert auf S. 22, 27, 55).

[Kru95] P. Kruchten. „The 4+1 View Model of architecture“. In: *IEEE Software* 12.6 (1995), S. 42–50. DOI: 10.1109/52.469759. URL: https://doi.org/10.1109%2F52.469759 (zitiert auf S. 25).

[LS80] B. P. Lientz, E. B. Swanson. *Software maintenance management*. Reading, MA: Addison-Wesley, 1980. URL: http://cds.cern.ch/record/101847 (zitiert auf S. 13).

[LS81] B. P. Lientz, E. B. Swanson. „Problems in Application Software Maintenance“. In: *Commun. ACM* 24 (1981), S. 763–769. ISSN: 0001-0782. DOI: 10.1145/358790.358796. URL: http://doi.acm.org/10.1145/358790.358796 (zitiert auf S. 14).

[Las02] N. Lassing. „Architecture-Level Modifiability Analysis“. In: *SIKS Dissertation* 2002 (2002), S. 1 (zitiert auf S. 36, 38, 89).

[Leh80] M. M. Lehman. „Programs, life cycles, and laws of software evolution“. In: *Proceedings of the IEEE* 68.9 (1980), S. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805 (zitiert auf S. 14).

[Leh96] M. M. Lehman. „Laws of software evolution revisited“. In: (1996), S. 108–124. DOI: 10.1007/bfb0017737. URL: http://dx.doi.org/10.1007/BFb0017737 (zitiert auf S. 14).

[Lem13] M. Lemcke. *Dynamische Prüfung von Spreadsheets*. 2013. DOI: 10.18419/opus-3128. URL: https://doi.org/10.18419/opus-3128 (zitiert auf S. 21, 27, 28, 55, 75).

[Lon+11] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, D. Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2011 (zitiert auf S. 45).

[Lud13] J. Ludewig. *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg: Dpunkt.verl, 2013. ISBN: 978-3-86490-092-1 (zitiert auf S. 14, 16, 17, 41, 51).

[Maa+14] W. Maalej, R. Tiarks, T. Roehm, R. Koschke. „On the Comprehension of Program Comprehension“. In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (2014), 31:1–31:37. ISSN: 1049-331X. DOI: 10.1145/2622669. URL: http://doi.acm.org/10.1145/2622669 (zitiert auf S. 20).

[Mar03] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003 (zitiert auf S. 19).

[NP90] J. T. Nosek, P. Palvia. „Software Maintenance Management: Changes in the Last Decade“. In: *Journal of Software Maintenance* 2.3 (1990), S. 157–174. ISSN: 1040-550X. DOI: 10.1002/smr.4360020303. URL: http://dx.doi.org/10.1002/smr.4360020303 (zitiert auf S. 14).

[Ope17] T. OpenGroup. *The SOA Source Book*. 2017. URL: http://www.opengroup.org/soa/source-book/soa/p1.htm#soa_definition (zitiert auf S. 19).

[PLB08] S. G. Powell, B. Lawson, K. R. Baker. „Impact of Errors in Operational Spreadsheets“. In: *CoRR* abs/0801.0715 (2008). URL: http://arxiv.org/abs/0801.0715 (zitiert auf S. 21).

[Pan98] R. R. Panko. „What We Know About Spreadsheet Errors“. In: *J. End User Comput.* 10.2 (Mai 1998), S. 15–21. ISSN: 1063-2239. URL: http://dl.acm.org/citation.cfm?id=287893.287899 (zitiert auf S. 21).

[Par72] D. L. Parnas. „On the Criteria to Be Used in Decomposing Systems into Modules“. In: *Commun. ACM* 15.12 (1972), S. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: http://doi.acm.org/10.1145/361598.361623 (zitiert auf S. 36).

[RA07] G. Rasool, N. Asif. „Software architecture recovery“. In: *Analysis* 321 (2007), S. 2740 (zitiert auf S. 24).

[RGK90] J. Radatz, A. Geraci, F. Katki. „IEEE standard glossary of software engineering terminology“. In: *IEEE Std* 610121990.121990 (1990), S. 3 (zitiert auf S. 21).

[Riv02] C. Riva. „Architecture Reconstruction in Practice“. In: *Software Architecture*. Springer Nature, 2002, S. 159–173. DOI: 10.1007/978-0-387-35607-5_10. URL: https://doi.org/10.1007%2F978-0-387-35607-5_10 (zitiert auf S. 20, 23–25).

[San+] N. Sangal, E. Jordan, V. Sinha, D. Jackson. „Using Dependency Models to Manage Complex Software Architecture“. In: *SIGPLAN Not.* 40.10 (), S. 167–176. ISSN: 0362-1340. DOI: 10.1145/1103845.1094824. URL: http://doi.acm.org/10.1145/1103845.1094824 (zitiert auf S. 49).

[Sch14] J. Scheurich. *Benutzerschnittstelle für einen Spreadsheet-Prüfstand*. 2014. DOI: 10.18419/opus-3276. URL: https://doi.org/10.18419/opus-3276 (zitiert auf S. 22, 27, 28).

[Sta84] T. A. Standish. „An Essay on Software Reuse“. In: *IEEE Trans. Softw. Eng.* 10.5 (1984), S. 494–497. ISSN: 0098-5589. DOI: 10.1109/TSE.1984.5010272. URL: http://dx.doi.org/10.1109/TSE.1984.5010272 (zitiert auf S. 23).

[Ste81] D. V. Steward. „The design structure system: A method for managing the design of complex systems“. In: *IEEE transactions on Engineering Management* 3 (1981), S. 71–74 (zitiert auf S. 49).

[Tot14] F. Toth. *Live-Prüfung von Spreadsheets während der Bearbeitung*. 2014. DOI: 10.18419/opus-3459. URL: https://doi.org/10.18419/opus-3459 (zitiert auf S. 22, 27).

[YM13] L. Yu, A. Mishra. „An empirical study of Lehmans law on software quality evolution“. In: *International Journal of Software & Informatics* 7.3 (2013), S. 469–481 (zitiert auf S. 14).

[Zit12] S. Zitzelsberger. *Fehlererkennung in Spreadsheets*. 2012. DOI: 10.18419/opus-2804. URL: https://doi.org/10.18419/opus-2804 (zitiert auf S. 21, 26–28, 53, 55, 57, 66, 69, 70).

Alle URLs wurden zuletzt am 01. 03. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift