HAW HAMBURG

BACHELORTHESIS
Jan Rehmeyer

Definition von Richtlinien für die Entwicklung von SAP Business One Erweiterungen zur Verbesserung der Softwarequalität

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jan Rehmeyer

Definition von Richtlinien für die Entwicklung von SAP Business One Erweiterungen zur Verbesserung der Softwarequalität

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Birgit Wendholt

Eingereicht am: 22. November 2019

Jan Rehmeyer

Thema der Arbeit

Definition von Richtlinien für die Entwicklung von SAP Business One Erweiterungen zur Verbesserung der Softwarequalität

Stichworte

SAP, Richtlinien, Softwarequalität, Quellcode, statische Analyse

Kurzzusammenfassung

Qualitativ hochwertige Produkte herzustellen und zu vertreiben gehört für beinahe jeden Betrieb zu einem der wichtigsten Ziele. Softwareentwicklung bildet dabei keine Ausnahme, doch nicht immer ist innerhalb eines Betriebes klar, was Qualität in diesem Zusammenhang bedeutet, oder wie diese erreicht werden kann. Das Ziel dieser Ausarbeitung ist es, eine Grundlage für die Einführung des Begriffs Softwarequalität innerhalb der Software-Entwicklungsabteilung eines Betriebes zu bilden.
Der Fokus liegt dabei auf Schwachstellen und Problemen innerhalb des Quellcodes von SAP Business One Erweiterungen.

Entwicklerspezifische Programmierungen sollen vereinheitlicht werden, um so den Quellcode klarer zu strukturieren und besser wartbar zu machen. Dazu wird der aktuelle Zustand des Betriebes anhand von Projekten der Abteilung analysiert und Maßnahmen und Richtlinien für erste Verbesserungen der Softwarequalität erstellt. Die beispielhafte Anwendung dieser Richtlinien auf ein Projekt zeigt, wie diese zu Verbesserungen in der Wartbarkeit und Lesbarkeit des Quellcodes führen. Für die Software-Entwicklungsabteilung ergibt sich aus der Anwendung der Ergebnisse dieser Ausarbeitung die Möglichkeit, einen ersten Schritt zur langfristigen Verbesserung von Softwarequalität zu machen.

Jan Rehmeyer

Title of Thesis

Defining Guidelines for the development of SAP Business One Extensions to improve Softwarequality

Keywords

SAP, Guidelines, Softwarequality, Source Code, static Analysis

Abstract

To create and sell high-quality products is one of the most important goals of almost every company. Software development is no exception, but quality in this context and how it can be achieved is not always clearly defined in a company. The goal of this thesis is to form a basis for the introduction of the term software quality within the software development department of a company. The focus is on vulnerabilities and problems within the source code of SAP Business One Extensions.

Developer specific programming should be unified to gain a clearer structure and better maintainability of the source code. To reach this goal the current state of the department is analyzed, based on projects of the department. Guidelines and actions for first improvements of software quality are created. The exemplary application of those guidelines on a project shows how they lead to improvements of the maintainability and readability of the source code. For the software development department the application of the results of this thesis results in the possibility to take a first step towards a long-term improvement of software quality.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Im Zuge eines dualen Studiums bei der Firma C & P Capeletti und Perl sollte eine praxisnahe und auf die Arbeitsweise des Betriebes zugeschnittene Ausarbeitung zu Aspekten der Softwareentwicklung erfolgen. Der Fokus liegt auf der Betrachtung der Qualität von hier entwickelten Softwareprojekten.

1.1 Über den Betrieb

Die C & P Capeletti & Perl ist ein Hamburger IT-Dienstleister, welcher seit über 30 Jahren sowohl die Bereitstellung von Hardware und Rechenzentrumsplätzen, als auch den Vertrieb, Support und das Anpassen von Softwarelösungen wie PDS oder SAP Business One anbietet. Kunden kommen vor allem aus dem Hamburger Mittelstand und sind zumeist im Handel tätig. [6]

Die einzelnen Softwarelösungen werden in separaten Abteilungen betreut. In jeder Abteilung arbeiten sowohl Vertriebler, als auch Berater und Entwickler eng zusammen. Für diese Ausarbeitung relevant ist die Abteilung der Anwendersoftware SAP Business One. Aktuell sind 3 Entwickler in der Abteilung tätig. Neben SAP Business One Erweiterungen entwickeln diese auch weitere Projekte, wie beispielsweise ein Kundenmanagementportal für den gesamten Betrieb, oder eine Lagerscanneranwendung, die unabhängig von SAP Business One vertrieben wird. Der Fokus der Abteilung liegt allerdings auf dem Support und der Erweiterung von SAP Business One.

1.2 Softwareentwicklung im Betrieb

Für die Softwareentwicklung innerhalb der SAP Business One Abteilung wurde noch kein übergreifendes Entwicklungskonzept erstellt und eingeführt, um die Arbeit an Produkten zu vereinheitlichen. Stattdessen wird sehr individuell entwickelt, bedingt zum einen durch unterschiedliche Wissensstände und Herangehensweisen der Entwickler, zum anderen durch fehlende Richtlinien für die einheitliche Entwicklung von Software und das Sicherstellen von Qualität.

Der Fokus wird während der Entwicklung stark auf eine schnelle Auslieferung und niedrige Entwicklungskosten gelegt. Aus diesem Grund werden häufig das ausführliche Dokumentieren, einheitliches und gut verständliches Softwaredesign oder sogar das Testen

der Software vernachlässigt. Vorteil davon ist, dass Kunden ihre Anforderungen schnell in Form einer Erweiterung für SAP Business One umgesetzt bekommen. Häufig kommt es allerdings zu anschließend notwendiger, längerer Betreuungszeit. In dieser muss versucht werden die, vom Kunden während des Einsatzes der Erweiterung entdeckten, Fehler zu beheben. Dies führt dazu, dass Kosten in Form von erneutem Zeitaufwand und auch weitere Entwicklungskosten anfallen, die entweder vom Betrieb selbst, oder aber vom Kunden getragen werden müssen und so zu Unzufriedenheit bei diesem führen können.

Ein Beispiel dafür ist die Anpassung einer Erweiterung im Bereich der Buchhaltung. Vor Auslieferung dieser Erweiterung wurde nicht getestet, welche Folgen ungültige Werte in bestimmten Berechnungen hatten. Dies führte dazu, dass nach einigen Wochen Meldungen vom Kunden kamen, dass sich Beträge, die eigentlich exakt gleich sein sollten, um große Summen unterscheiden. Durch manuelles Testen konnte nach mehreren Arbeitsstunden herausgefunden werden, welcher Eintrag den Fehler verursacht und durch eine statische Prüfung im Quellcode abgefangen werden. Hier wären erheblicher Zeitaufwand und anfallende Kosten vermeidbar gewesen, wenn schon bei der Entwicklung oder während der Entwurfsphase dieser Erweiterung genauer getestet worden wäre. Zudem ist der Quellcode durch diese schnelle, unsaubere Problemlösung unverständlicher geworden.

In Zukunft sollen auch größere Projekte, sowohl intern, als auch für Kunden umgesetzt werden. So sind beispielsweise das bereits angesprochene Kundenmanagementportal oder die Lagerscanneranwendung in Planung oder bereits in Arbeit. Diese gestalten sich weitaus komplexer als einzelne Kundenerweiterungen. Mit steigender Projektgröße und höherer Komplexität steigen die Möglichkeiten, dass Fehler bei der Implementierung gemacht werden.

1.3 Motivation und Problemstellung

Um zu vermeiden, dass in zukünftigen Projekten ähnliche Fehler entstehen, wie in Kapitel 1.2 auf Seite 1 beschrieben, und um die Flexibilität innerhalb des Entwicklerteams zu verbessern ist es sinnvoll, Vereinheitlichungen in der Entwurfs- und Implementationsphase zu schaffen. Damit sollen alle Entwickler auf einen Stand gebracht werden, um einen Firmenstandard für Softwareentwicklung herzustellen.

Eine Schwierigkeit ist, dass viele Prozesse und Techniken entwicklerspezifisch durchgeführt werden und so dazu führen, dass die Übertragbarkeit von Entwicklern auf andere Projekte eingeschränkt oder erschwert wird. Beispiele dafür sind die Namensgebung oder die Dokumentation von Quellcode. Durch zahlreiche schnelle und unsaubere Ausbesserungen an vielen Stellen in den Projekten, wird der Quellcode langfristig schwerer zu verstehen oder zu analysieren. Am Beispiel aus Kapitel 1.2 auf Seite 1 zeigt sich, dass

die eingeschobene Lösung das initiale Problem zwar behoben, jedoch weitere, schwer zu wartende Probleme, verursacht, hat.

Das Umsetzen von Modellen und Theorien zur Verbesserung der Softwarequalität kann für einen Betrieb mit kleiner Entwicklungsabteilung sinnvoll sein, um die Entwicklungsdauer und langfristig anfallende Wartungskosten zu senken, sowie die Kundenzufriedenheit durch bessere Produkte zu erhöhen.

Das Ziel dieser Ausarbeitung ist es, Maßnahmen für die Verbesserung der Softwarequalität in Form von Richtlinien für die Entwicklung von SAP Business One Erweiterungen innerhalb der SAP Business One Abteilung zu schaffen. Das Ergebnis bildet den ersten Schritt eines Konzeptes zur langfristigen Qualitätssicherung. Dies geschieht durch die Einführung von einfachen und grundlegenden Maßnahmen und Richtlinien innerhalb der Softwareentwicklung der Abteilung. Diese werden dazu beitragen, durch geringen Entwicklungsaufwand Kosten zu sparen und durch das Erstellen qualitativ hochwertigerer Software die Kunden zufriedener zu stellen.

1.4 Aufbau der Arbeit

Die Ausarbeitung gliedert sich neben dieser Einleitung in folgende Kapitel:

Grundlagen

In den Grundlagen werden Fachwörter und Inhalte beschrieben, die für das Verständnis dieser Ausarbeitung hilfreich sind.

Verbesserungsansatz

In diesem Kapitel wird ein Ansatz für den generellen Aufbau dieser Ausarbeitung sowie für die Durchführung der Verbesserungen innerhalb der Abteilung erarbeitet.

Umfrage

In diesem Kapitel wird der aktuelle Zustand und die allgemeine Einstellung der Abteilung in Bezug auf Softwarequalität festgestellt.

Ziele

Das Kapitel Ziele fasst umsetzbare Qualitätsziele für die Abteilung zusammen.

Analyse

In der Analyse werden, auf Basis der gewählten Ziele sowohl Merkmale, als auch Metriken und Werkzeuge betrachtet. Unter Einsatz dieser werden drei SAP Business One Erweiterungen aus der Abteilung analysiert, um Schwachstellen und Verbesserungsmöglichkeiten festzustellen.

Beschreibung der Richtlinien

In diesem Kapitel werden die, auf Basis der Analyse-Ergebnisse erstellten Richtlinien, die für die Verbesserung der Softwarequalität genutzt werden können, zusammengefasst und beschrieben. Anschließend wird geplant, wie diese Richtlinien innerhalb der Abteilung umgesetzt werden können.

Umsetzung der Richtlinien

In diesem Kapitel wird eine beispielhafte Umsetzung der definierten Richtlinien durchgeführt. Dazu wird eine der bereits analysierten Erweiterungen auf Basis der Richtlinien refaktoriert und anschließend mit dem vorherigen Stand verglichen.

Auswertung

Den letzten Teil dieser Ausarbeitung bildet ein Überblick der erreichten Ergebnisse. Darauf folgen Überlegungen zu den nächsten Schritten in der Qualitätsverbesserung des Betriebes.

2 Grundlagen

2.1 Enterprise Resource Planning

Enterprise Resource Planning (kurz ERP) steht für Programme, die sich mit der Planung und der Steuerung von Geschäftsprozessen von Betrieben befassen. Beispiele für solche Prozesse sind der Ein- und Verkauf, oder die Lagerführung. [7]

2.2 SAP Business One

SAP Business One (abgekürzt SAP B1) ist ein ERP-System für kleine und mittelständische Unternehmen und stellt Funktionen zur Verwaltung der Geschäftsprozesse eines Betriebes zur Verfügung. [25]

2.3 Erweiterung

Erweiterungen sind Programme, die zusätzliche Funktionalitäten, Darstellungen oder Berechnungen für SAP B1 anbieten. Erweiterungen aus der Entwicklung von C & P greifen entweder über das Coresuite Framework oder über das SAPBusinessOneSDK auf Objekte des SAP B1 Standards zu und können diese auslesen, verändern oder sogar löschen. Im Folgenden sind mit Erweiterungen, wenn nicht anders beschrieben, SAP B1 Erweiterungen gemeint.

2.4 Coresuite Framework

Das Coresuite Framework, entwickelt von der Schweizer Softwarefirma Coresystems, ist Teil einer Erweiterung (Coresuite Country Package) für SAP B1. Es bietet unter anderem eine Schnittstelle zu SAP B1 an, die von der SAP Business One Abteilung genutzt wird, um die Verbindung zwischen weiteren SAP B1 Erweiterung und einer SAP B1 Instanz herzustellen. [3]

2.5 Beleg

Ein Beleg in SAP B1 enthält die notwendigen Informationen um Teile eines Waren- oder Handelsgeschäfts abwickeln zu können. Dazu gehören beispielsweise betreffende Geschäftspartner, Datums- und Logistikinformationen und verkaufte Artikel. Beispiele für Belege sind Angebote, Ein- und Ausgangsrechnungen oder Retouren.

2.6 Verknüpfungsplan

Belege in SAP B1 können entweder auf einem Vorgängerbeleg basieren oder selbst als Basis für Folgebelege dienen. Diese Zusammenhänge können in einem sogenannten Verknüpfungsplan graphisch dargestellt werden.

2.7 Demoumgebung

Um Erweiterungen oder Konfigurationen an SAP B1 für einzelne Kunden testen zu können, werden sogenannte Demoumgebungen von der Abteilung eingesetzt. Diese bestehen aus passend konfigurierten Systemen und Kopien der Daten der einzelnen Kunden, um deren Systeme möglichst akkurat wiederzugeben. Da beinahe jeder Kunde unterschiedliche SAP B1 Versionen, Erweiterungen und Einstellungen nutzt, muss für jeden unterschiedlichen Kunden eine eigene Demoumgebung vorliegen, um diesen repräsentieren zu können.

2.8 SAPbobsCOM und SAPbouiCOM

Diese beiden Bibliotheken bilden die Schnittstellen zu SAP B1 Instanzen und stellen Funktionen für den Zugriff von Objekten und Funktionen aus dem SAP B1 Standard bereit. SAPbobsCOM ist für Daten, SAPbouiCOM für UI-Elemente, zuständig.

2.9 Magisches Dreieck der Softwareentwicklung

In der Entwicklung von Softwareprojekten gibt es laut dem Modell des Magischen Dreiecks [5] drei wesentliche Aspekte, die unterschiedlich stark priorisiert werden können und Einfluss auf die jeweils anderen Aspekte haben. Die Aspekte sind Qualität, Kosten und Entwicklungsdauer des Produktes und geben an, wo bei der Entwicklung eines Projektes der Fokus liegt, beziehungsweise liegen sollte. Abbildung A.1 auf Seite 71 im Anhang stellt dies grafisch dar.

2.10 Namensschemata in C#

In C# kommen zwei wesentliche Benennungssysteme zum Einsatz um die Groß- und Kleinschreibung festzulegen [16].

Zum einen wird camelCase genutzt, um Parameter und lokale Variablen in Methoden zu bezeichnen, dabei wird das Wort klein geschrieben, bei zusammengesetzten Wörtern wie beispielsweise „camelcase“ wird der Anfangsbuchstabe des angehängten Wortes groß geschrieben.

Zum anderen wird Pascalcase für alle anderen Bezeichner genutzt und unterscheidet sich von camelCase dadurch, dass der erste Buchstabe auch groß geschrieben wird, wie beispielsweise in „PascalCase“ zu erkennen ist.

2.11 Glossar

An dieser Stelle werden weitere Fachbegriffe kurz zusammengefasst und beschrieben.

.NET-Richtlinien	Von Microsoft herausgegebene Software-Plattform
Access Modifier	Kennworte für den Zugriff auf Objekte innerhalb eines Programms.
as-is-utility	Merkmale aus Boehmes Softwaremodell, die Effizienz, Verlässlichkeit und Übertragbarkeit eines Programms beschreiben
Bug	Fehlverhalten eines Computerprogramms
C#	Programmiersprache von Microsoft, angewandte Programmiersprache für die Entwicklung von SAP B1 Erweiterungen bei C & P
camelCase	Namenskonvention für die Groß- und Kleinschreibung von Bezeichnern. Hierbei wird das Wort klein geschrieben, weitere angehängte Teilworte beginnen mit einem Großbuchstaben.
Capability Maturity Model	Modell zur Qualitätsmessung und -Verbesserung von Software
Checkstyle	Werkzeug für statische Quellcodeüberprüfung
Code Smell	Funktionierender Programmcode mit schlechter Struktur oder Darstellung (von SonarQube verwendet)
Exception	Ausnahme im Programmablauf, kann für strukturierte Fehlerbehandlung genutzt werden
GNU Lesser General Public License	Von der Free Software Foundation entwickelte Lizenz für freie Software
Konkatenation	Verkettung von Zeichen oder Zeichenketten

Line-kommentar	Einzeiliger Kommentar innerhalb des Quellcodes, in C# mit // gekennzeichnet.
Maintainability	Kategorie von Merkmalen aus Boehmes Qualitätsmodell. Beschreibt die Lesbarkeit und Änderbarkeit eines Programms.
Namespace	Pfadangabe von Namen in einer Programmstruktur in C#
PascalCase	Namenskonvention für die Groß- und Kleinschreibung von Bezeichnern. Hierbei werden jeweilige Teilworte mit großem Anfangsbuchstaben geschrieben.
Quality Improvement Paradigm	Abgekürzt QIP, eine Methodik zur Verbesserung von Produkt- und Prozessqualität
Refaktorierung	Strukturverbesserung eines Programms unter Beibehaltung des Programmablaufs
Repository	Projektarchiv in der eingesetzten Versionskontrolle
SAPBusinessOneSDK	Sammlung von Werkzeugen und Bibliotheken für die Entwicklung von SAP Business One Software
snake_case	Namenskonvention für die Groß- und Kleinschreibung von Bezeichnern. Hierbei werden Teilworte klein geschrieben und mit einem Unterstrich verbunden.
SQL-Abfrage	Befehl innerhalb der Datenbanksprache SQL
Statische Analyse	Analyse von Quellcode ohne diesen auszuführen
Stringbuilder	Stellt eine Klasse für die Darstellung und Veränderung von Strings dar
Technische Schuld	Benötigte Zeit, Code Smells zu beheben (von SonarQube verwendet)
UI	User-Interface, die Benutzeroberfläche eines Programms
Ungarische Notation	Namenskonvention für die Benennung von Bezeichnern [20]
User Manual	Benutzerhandbuch, beinhaltet die Richtlinien aus dieser Ausarbeitung
Visual Studio	Entwicklungsumgebung für .NET-Entwicklung, hergestellt von Microsoft. Wird in der Abteilung für die Entwicklung von SAP Business One in Version 2017 genutzt.

3 Verbesserungsansatz

Nach Liggesmeyer [8] gibt es zwei wesentliche Ansätze zur Verbesserung der Softwarequalität innerhalb eines Betriebes. Diese unterscheiden sich effektiv in der Methodik der Durchführung. Zum einen wird dabei der modellbasierte Ansatz genannt, dem gegenüber steht der kontinuierliche oder auch iterative Ansatz. Modellbasierte Ansätze nutzen statische, unveränderliche Referenzmodelle, an die sich angenähert werden soll, während kontinuierliche Ansätze in mehreren Zyklen individuelle Lösungen entwickeln. In dieser Ausarbeitung wird eine Kombination aus beiden Ansätzen verwendet.

3.1 Kontinuierliche Ansätze

Liggesmeyer nennt in seinem Buch Software Qualität verschiedene kontinuierliche Ansätze, wie beispielsweise Plan-Do-Check-Act, Quality Improvement Paradigm oder Six Sigma [10]. Einer davon, das Qualitäts-Verbesserungs-Paradigma[1] zeigt dabei Ideen und Schritte auf, die für diese Ausarbeitung interessant und für die Nutzung in der Abteilung sinnvoll scheinen.

Beim QIP werden dazu sechs Schritte durchgeführt, um eine Basis zu schaffen, von der aus bei Betrachtung von Softwareproblemen in der nächsten Iteration gestartet werden kann. Die Schritte des QIP sind in der Tabelle A.1 auf Seite 71 im Anhang zu sehen.

3.2 Modellbasierte Ansätze

Modellbasierte Ansätze eignen sich besonders zum Einsatz, wenn noch keine eigenen Erfahrungen in einem Bereich, wie beispielsweise der Qualitätssicherung, in einem Betrieb vorhanden sind. Durch den Vergleich mit bestehenden Referenzmodellen lassen sich besonders die Identifikation und Klassifizierung von relevanten Aspekten unterstützen. Im Gegensatz zum kontinuierlichen Ansatz ist das Beheben von konkreten Problemen mit modellbasierten Ansätzen zwar auch möglich, aber aufgrund der oft sehr speziellen Anforderungen und Eigenschaften eines Betriebes nicht immer einfach, ein passendes Modell zu finden.

Modellbasierte Ansätze sind beispielsweise ISO 15504, das Capability Maturity Model oder BOOTSTRAP/Koch, diese unterscheiden sich hauptsächlich in der Behandlung

[1] Quality Improvement Paradigm, kurz QIP

von Merkmalen, Prozessen, oder Arbeitsweisen und orientieren sich oftmals an Erfahrungen bereits etablierter Unternehmen [11].

3.3 Zusammengesetzter Ansatz für den Betrieb

Wie von Liggesmeyer beschrieben, eignen sich modellbasierte Ansätze gut zur groben Identifikation und Analyse, während sich kontinuierliche Ansätze eher für das Entwickeln von Verbesserungsansätzen und die Implementation von konkreten Optimierungen anbieten. Weiterhin führt Liggesmeyer auf, dass die verschiedenen Ansätze nicht komplett getrennt genutzt werden müssen, sondern sich im Gegenteil sogar ergänzen und erweitern.[9]

Auf Basis dessen wurde daher, aufbauend auf der Struktur des QIP, eine kombinierte Methodik entwickelt. In Tabelle 3.1 auf Seite 10 sind die Schritte und die durch diese zu erreichenden Ziele dargestellt.

Zusätzlich ist in der Tabelle die Struktur dieser Ausarbeitung zu erkennen, jeder Schritt wird in einem der angegebenen Kapitel behandelt. Die Modelle des ISO 9126, sowie das Softwarequalitätsmodell Boehmes werden im Folgenden als Grundlage der Qualitätsmerkmale für die Definition und die Auswahl geeigneter Prozesse und Methoden genutzt. Dies wird in Kapitel 6.1 auf Seite 23 genauer beschrieben.

Der zusammengesetzte Ansatz nutzt also Aspekte aus dem kontinuierlichen Ansatz QIP für die Vorgehensweise der Verbesserungsmaßnahmen, sowie Aspekte aus dem modellbasierten Ansatz für die Orientierung und Einordnung der zu behebenden Schwachstellen innerhalb der Abteilung.

Schritt	**Aktion**	**Kapitel**
1	Charakterisieren und Verstehen der Organisation und Umgebung unter Einbezug existierender Informationen und Modelle	4
2	Definition von umsetzbaren Verbesserungszielen	5
3	Auswahl geeigneter Prozesse, Methoden, Techniken und Werkzeuge zur Umsetzung der Verbesserungsziele auf Basis der gewählten Modelle	6
4	Festhalten von Maßnahmen in Form von Richtlinien	7
5	Umsetzung der Richtlinien, Analyse der Resultate, Aufzeigen der Schwächen und Verbesserungsempfehlungen	8
6	Konsolidierung der gemachten Erfahrungen und Bildung einer Basis für weitere Iterationen	9

Tabelle 3.1: Schritte im zusammengesetzten Ansatz

Diese Ausarbeitung ist als Durchführung der ersten Iteration dieser gewählten Methodik zu verstehen. Sie endet mit der Konsolidierung des erarbeiteten Wissens und der gemachten Erfahrungen in Form der zu erstellenden Richtlinien in dem Ergebnis dieser Ausarbeitung. Darauf folgende Iterationen können anschließend innerhalb der Abteilung besprochen und auf Basis dieser Ausarbeitung durchgeführt werden, um komplexere oder speziellere Themen abzudecken. Diese werden im Kapitel 9.2: Nächste Iterationen und Ausblick auf Seite 64 genauer beschrieben.

4 Umfrage

Die im Folgenden beschriebene Umfrage wird in der SAP Business Abteilung des Betriebes durchgeführt, um eine Basis für die in dieser Ausarbeitung relevanten Aspekte der Softwarequalität zu erhalten. Befragt werden Vertriebler, Entwickler und Berater aus der Abteilung. Das Ziel ist, den in Kapitel 3.3 auf Seite 10 beschriebenen, ersten Schritt des zusammengesetzten Ansatzes durchzuführen. Aussagen und Einschätzungen der für die Entwicklung innerhalb der Abteilung verantwortlichen Mitarbeiter werden festgehalten und bilden ein Grundverständnis für die darauf folgenden Kapitel. Insgesamt haben neun Personen an der Umfrage teilgenommen.

4.1 Frage 1

Um festzustellen, welche der drei Aspekte des Magischen Dreiecks der Softwareentwicklung[1] den verantwortlichen Mitarbeitern am wichtigsten sind, sollen in der ersten Frage zehn Punkte auf die unten angegebenen Optionen verteilt werden.

- Hohe Produktqualität
- Geringe Entwicklungskosten
- Kurze Entwicklungsdauer

Aus dem daraus entstehenden arithmetischen Mittel wird sich versprochen, einen Einblick in die allgemeine Einstellung zu den drei Parametern Qualität, Kosten und Entwicklungsdauer zu erhalten um dies in der Analyse mit der aktuellen realen Umsetzung zu vergleichen. Es ergibt sich der folgende Durchschnitt an Punkteverteilungen:

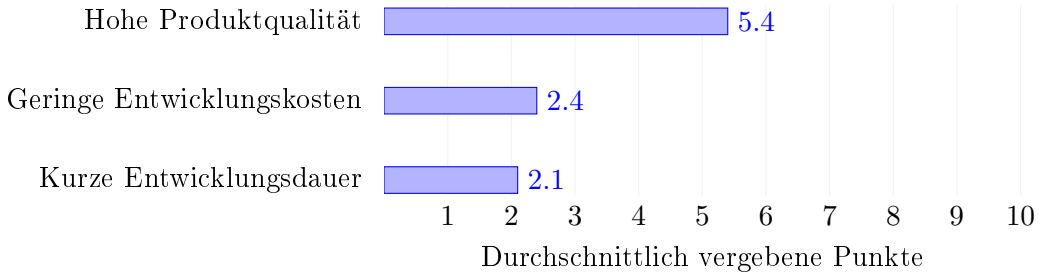

Abbildung 4.1: Antworten zu Frage 1

[1] siehe dazu Kapitel 2.9 auf Seite 6

Abbildung 4.1 auf Seite 13 stellt die Ergebnisse der Umfrage dar. Diese zeigen, dass hohe Qualität eines Produktes durchschnittlich circa doppelt so wichtig erachtet wird, wie eine kurze Entwicklungszeit und niedrige Entwicklungskosten. Das Ergebnis dieser Frage zeigt, dass innerhalb der Abteilung der Wunsch existiert, die Qualität der entwickelten Produkte an erste Stelle zu setzen.

4.2 Frage 2

Mit dieser Frage soll herausgefunden werden, welche Art von Projekten sich die befragten Mitarbeiter als Hauptteil der Softwareentwicklung der Abteilung vorstellen. Sie werden gebeten, anzugeben, ob der Fokus auf größere, standardisierte Projekte, oder auf kleinere, individuelle Erweiterungen gelegt werden sollte. Dabei ist eine Angabe von 0 die ausschließliche Fokussierung auf standardisierte Projekte, eine Angabe von 100 dagegen die ausschließliche Fokussierung auf kleinere individuelle Erweiterungen.

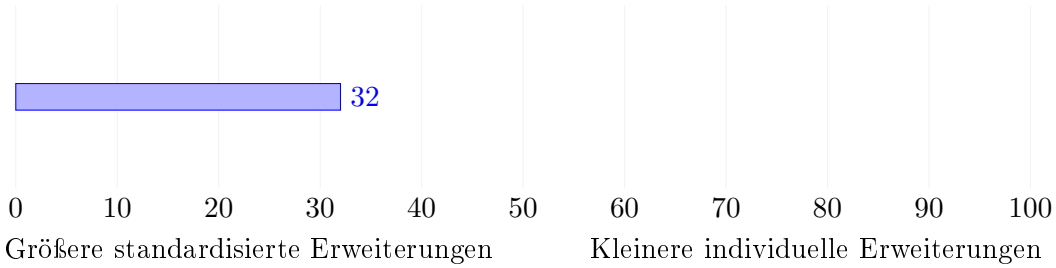

Abbildung 4.2: Antworten zu Frage 2

Die Antworten dieser Frage sind in Abbildung 4.2 auf Seite 14 dargestellt. Sie zeigen, dass der Wunsch innerhalb der Abteilung besteht, den zukünftigen Fokus durchschnittlich zu circa zwei Dritteln auf größere, standardisierte Projekte zu legen. Die Ursachen hierfür könnten beispielsweise der Wunsch nach weniger unstrukturierten Projekten, oder durch Richtlinien vereinheitlichten Projekten sein.

4.3 Frage 3

In der dritten Frage soll festgestellt werden, welcher Anteil der Entwicklungsdauer aufgewendet werden sollte, um qualitativ hochwertige Software herzustellen. Dafür werden die Befragten gebeten, anzugeben, welcher Anteil der gesamten Entwicklungszeit einer Erweiterung für Qualitätssicherung[2] aufgewendet werden sollte. Die Antworten werden dabei als Prozentsatz der gesamten Entwicklungszeit angegeben.

[2]Wie beispielsweise das Testen oder Dokumentieren von Quellcode

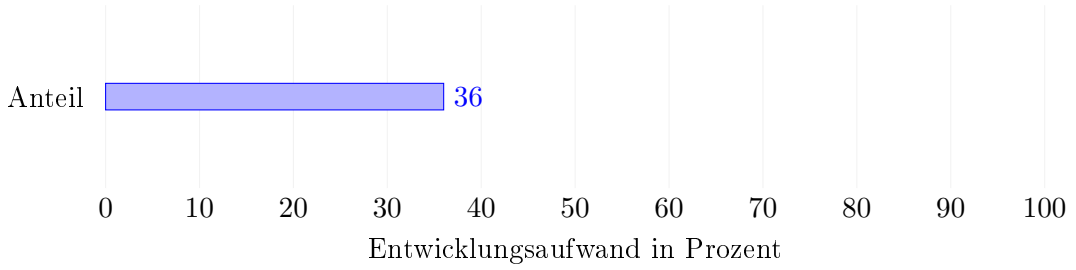

Abbildung 4.3: Antworten zu Frage 3

In den Antworten von Frage 3, zu sehen in Abbildung 4.3 auf Seite 15, ist zu erkennen, dass nach Meinung der Befragten durchschnittlich ein Anteil von 36% genannt wird. Dies entspricht ungefähr einem Drittel des gesamten Entwicklungsaufwandes eines Projektes, der für die Verbesserung von qualitativen Aspekten der Software genutzt werden sollte.

4.4 Frage 4

In dieser Frage werden die befragten Mitarbeiter gebeten, auf einer Skala von 0 bis 100 anzugeben, wo sie den Schwerpunkt für die Entwicklung von Projekten legen würden. 0 auf der Skala entspricht dabei dem schnellen und potentiell schlechter wartbarem Einbauen von neuen Features / Anpassungen. 100 auf der Skala hingegen steht für eine langsamere, aber dafür potentiell gründlichere und besser dokumentierte Entwicklung, die zu langfristiger Wartbarkeit führen kann.

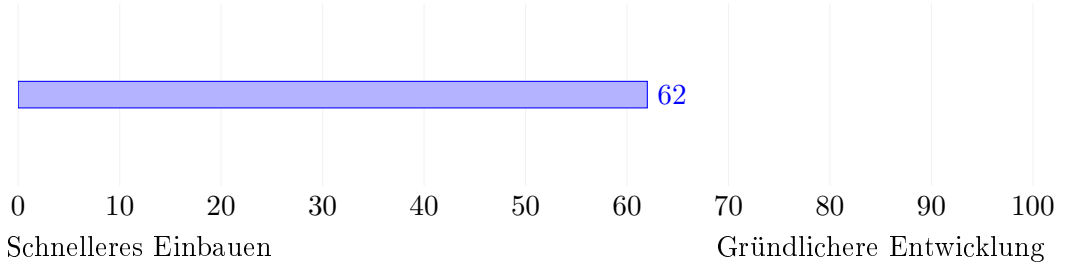

Abbildung 4.4: Antworten zu Frage 4

Ähnlich wie auch in Frage 3 ist hier zu erkennen, dass die Befragten durchschnittlich eher dafür sind, längere Entwicklungszeiten in Kauf zu nehmen, um dafür langfristig wartbare und besser verständliche Software zu entwickeln, anstatt den Fokus auf möglichst kurze Entwicklungszeiten und schnelle Anpassungen zu legen. Abbildung 4.4 auf Seite 15 zeigt dies.

4.5 Frage 5

Diese Frage wird gestellt, um herauszufinden, welche Merkmale von Softwarequalität innerhalb der Abteilung aktuell als besonders wichtig oder weniger wichtig erachtet werden. Dazu wird das Qualitätsmodell des ISO 9126[3] als Grundlage genutzt. Die befragten Mitarbeiter sollen die folgenden Merkmale nach Wichtigkeit sortieren:

- **Funktionalität** (Erfüllt das Programm die geforderten Funktionen)
- **Zuverlässigkeit** (Wie stabil und erwartungsgemäß läuft das Programm)
- **Benutzbarkeit** (Welchen Aufwand erfordert die Benutzung des Programms)
- **Effizienz** (Was leistet das Programm mit welchen Ressourcen)
- **Änderbarkeit** (Welchen Aufwand erfordern Änderungen am Programm)
- **Übertragbarkeit** (Lassen sich Aspekte des Programms auf andere Anwendungen oder Systeme übertragen)

In Abbildung 4.5 auf Seite 16 ist der Durchschnitt der Antworten zu sehen. Je höher eines der Merkmale bei einem einzelnen Mitarbeiter einsortiert ist, umso mehr Punkte werden diesem zugewiesen. So erhält ein Merkmal an erster Stelle fünf Punkte, während ein Merkmal an letzter Stelle null Punkte erhält. Diese Punkte werden von allen Beantwortungen zusammengerechnet und geteilt durch die Anzahl der Antworten, um einen Durchschnitt zu erhalten. Eine hohe Punktzahl in der Grafik ist also äquivalent mit einer häufigen Einordnung des Merkmals an hoher Priorität.

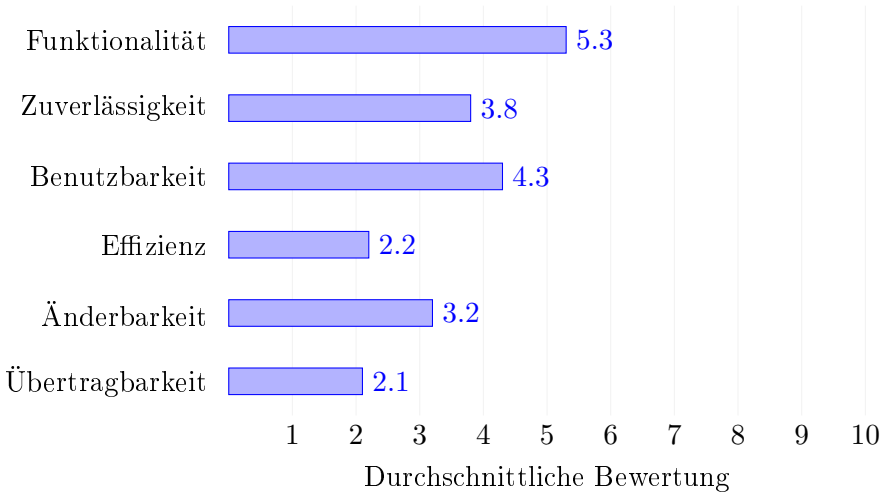

Abbildung 4.5: Antworten zu Frage 5

Im Durchschnitt der Auswertung der Antworten, dargestellt in Abbildung 4.5 auf Seite 16, ist zu sehen, dass hohe Funktionalität und gute Benutzbarkeit der Erweiterungen

[3] Siehe dazu auch Kapitel 6.1 auf Seite 23

im Vordergrund stehen. Die Änderbarkeit und Übertragbarkeit von Projekten hingegen werden im Durchschnitt als weniger wichtig erachtet.

4.6 Frage 6

In Frage 6 werden wesentliche Merkmale von SAP B1 Erweiterungen aufgeführt. Diese decken die wichtigsten Bereiche der Softwareentwicklung für SAP B1 Erweiterungen ab und sind aus mehreren Qualitätsmodellen[4] zusammengefasst abgeleitet. Die Befragten Mitarbeiter sollen diese so sortieren, wie sie ihrer Einschätzung nach zum Zeitpunkt der Umfrage in Erweiterungen zu finden sind.

- **Einfachheit** der Bedienung
- **Sicherheit** und Fehlertoleranz
- **Dokumentation** der Erweiterung
- **Anpassbarkeit** und Flexibilität
- **Wartbarkeit** und **Lesbarkeit** des Codes

Wie in der vorherigen Frage gilt auch hier, dass Antworten mit hohen Punktzahlen von den Mitarbeitern häufiger als wichtig eingestuft werden.

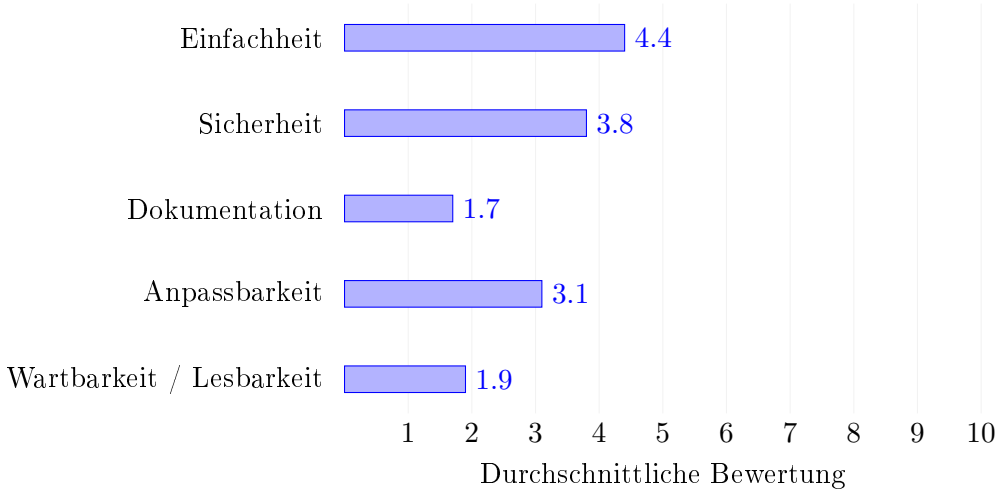

Abbildung 4.6: Antworten zu Frage 6

Im Durchschnitt der Antworten in Abbildung 4.6 auf Seite 17 ist deutlich zu erkennen, dass sowohl Dokumentation von Quellcode, als auch die damit verbundene Wartbarkeit und Lesbarkeit von den Mitarbeitern durchschnittlich als schlechter gepflegt eingestuft werden, als die Sicherheit und die Bedienung einer Erweiterung.

[4] ISO 9126 sowie Boehmes Qualitätsmodell

4.7 Frage 7

Um eine Gegenüberstellung des Ist-Zustandes (aus Frage 6) mit dem Soll-Zustand zu erhalten, werden die Befragten in dieser Frage erneut gebeten, die Merkmale nach Relevanz zu sortieren. Diesmal allerdings so, wie sie im Optimalfall in Erweiterungen zu finden sein sollten.

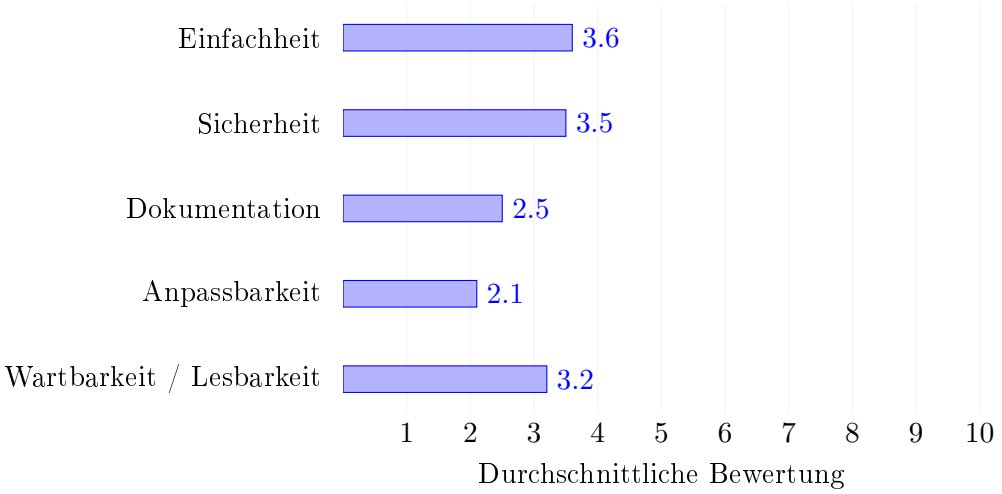

Abbildung 4.7: Antworten zu Frage 7

Abbildung 4.7 auf Seite 18 zeigt die Ergebnisse dieser Frage. Im Vergleich mit den Antworten aus Frage 6 sind Dokumentation und Wartbarkeit durchschnittlich circa 40% wichtiger eingestuft worden, als sie es zur Zeit sind. Die Einfachheit der Bedienung, sowie die Anpassbarkeit des Codes sind dafür als weniger relevant eingestuft worden, während die Sicherheit der Erweiterungen laut den Befragten gleichbleibend wichtig sein soll.

4.8 Auswertung der Umfrage

Die Ergebnisse der Umfrage zeigen, dass innerhalb der Abteilung der Wunsch besteht, einen stärkeren Anspruch an die Qualität der entwickelten Projekte zu stellen und dafür zu sorgen, langfristig besseren Quellcode zu produzieren. Besonders die Verbesserung von Wart- und Lesbarkeit des Quellcodes, sowie die Dokumentation der Projekte wurden als Schwerpunkte genannt.

Auch zeigt sich durch die Umfrage der Wunsch nach stärkerer Konformität und Einheitlichkeit der Projekte. Besonders im Hinblick auf zukünftige, größere Projekte ist ein Standard in der Entwicklung, sowie Vereinheitlichung von Aspekten der Entwicklung als relevant erachtet worden.

Eine Verbesserung der Dokumentation von SAP B1 Erweiterungen innerhalb der Entwicklungsabteilung wird als ein wichtiger Teil des Verbesserungsbedarfes genannt und sollte dementsprechend in der folgenden Analyse betrachtet werden.

Zusammenfassend ergibt sich ein angestrebter Fokus auf die Verbesserung der Leserlichkeit und damit verbundenen Wartbarkeit des Quellcodes. Vereinheitlichungen von Konventionen, Prozessen und Arbeitsweisen sowie eine Erhöhung der Softwarequalität durch das Einführen eines einheitlichen Konzeptes für die Dokumentation von SAP B1 Erweiterungen sind gewünscht. In den folgenden Kapiteln werden daher hauptsächlich Ziele, Merkmale und Metriken betrachtet, die Aussagekraft über diese Aspekte besitzen.

5 Definition von Verbesserungszielen

Die Umfrage aus Kapitel 4 auf Seite 13 konnte bereits erste Einblicke in die Einstellung der Abteilung zu Aspekten der Softwarequalität liefern. Diese werden in diesem Kapitel als konkrete, umsetzbare Ziele ausformuliert.

Aus weiteren Gesprächen und Diskussionen mit den Entwicklern und Beratern der Abteilung, sowie aus immer wiederkehrenden Problemen während der Entwicklung, konnten allgemeine Ziele erarbeitet werden. Eine ausführliche und erschöpfende Qualitätszielanalyse innerhalb der Abteilung würde den Rahmen dieser Ausarbeitung sprengen, da eine korrekte Anwendung von Methoden[1] dazu an vielen Stellen noch nicht sinnvoll ist. So sind die Ziele beispielsweise nicht terminiert oder exakt messbar, es geht eher darum, den Begriff der Qualitätsziele in die Abteilung einzuführen und vorerst nur grobe Ziele festzuhalten.

Folgende Ziele wurden erarbeitet:

- Einarbeitungszeit von Entwicklern in Projekte minimieren, um Produktivität zu verbessern.
- Softwareprojekte vereinheitlichen, um flexible Bearbeitung durch unterschiedliche Entwickler zu ermöglichen.
- Bessere Dokumentation des Quellcodes, um diesen verständlicher zu machen.
- Wissen übertragbar machen, um zu ermöglichen, dass ein Entwickler die Aufgaben von anderen Entwicklern ohne Probleme übernehmen kann.
- Aufnahme der Anforderungen verbessern und die Probleme, die eine Erweiterung lösen soll, klarer definieren, um häufige Nachbesserungen zu vermeiden.
- Software ausreichend testen, um Fehler während der Entwicklung zu beheben.

Festzuhalten ist, dass diese Ziele, sowohl im Rahmen dieser Ausarbeitung, als auch innerhalb der Abteilung umgesetzt und verfolgt werden. Die Ziele sind aufgrund der fehlenden konkreten Aspekte nur schwer zu überprüfen, weshalb im folgenden Teil mehrere Modelle und Werkzeuge eingesetzt werden, die diese Ziele komplett oder teilweise abdecken und beschreiben können[2]. Zukünftige Iterationen können unter Einsatz der

[1] Beispielsweise der SMART-Methodik [29]
[2] Siehe dazu auch Kapitel 6 auf Seite 23

gemachten Erfahrungen dieser Ausarbeitung genauere Ziele definieren, um speziellere Teilbereiche der Softwarequalität abzudecken.

6 Analyse

Auf Basis der festgelegten Ziele werden in diesem Kapitel zuerst zu betrachtende Qualitätsmerkmale und Metriken anhand von Referenzmodellen erarbeitet. Unter Einsatz von, in Kapitel 6.3 auf Seite 25, festgelegten Werkzeugen werden anschließend drei SAP Business One Erweiterungen aus der Entwicklung der Abteilung ausgewählt und auf die festgelegten Gesichtspunkte hin analysiert.

6.1 Qualitätsmerkmale

In diesem Kapitel wird erarbeitet, welche Merkmale von Softwarequalität für die Analyse der Projekte in Kapitel 6.4 auf Seite 26 relevant sind. Diese werden in der folgenden Analyse eingesetzt, um die in Kapitel 5 auf Seite 21 festgelegten Ziele zu erreichen.

Wie bereits in Kapitel 3.3: Zusammengesetzter Ansatz für den Betrieb auf Seite 10 erklärt wurde, sollen vor allem für die ersten Schritte, der in dieser Ausarbeitung beschriebenen Iteration, einfache Modelle gewählt werden, an denen sich die Analyse und die Maßnahmen zur Qualitätsverbesserung orientieren sollen. Die Qualitätsmodelle des ISO 9126 [1], sowie Boehmes Qualitätsmodell [2] wurden aufgrund ihrer Einfachheit und leicht zu verstehenden Merkmale gewählt, um als Grundlage für die Untersuchung der in dieser Ausarbeitung genannten Projekte genutzt zu werden.

ISO 9126 definiert Kriterien für die Qualitätsmessung von Software als Produkt. Das bedeutet, dass Prozesse und Methoden der Entwicklung nicht, oder nur teilweise betrachtet werden. Stattdessen bietet ISO 9126 Merkmale für externe und interne Qualität des Quellcodes. Externe Qualität beschreibt dabei Merkmale, die bei dem Ausführen des Programms auftreten, während interne Merkmale vor allem für die Betrachtung des Quellcodes relevant sind. Diese können untersucht werden, ohne den Quellcode ausführen zu müssen. Besonders die Betrachtung der internen Merkmale ist daher für die Analyse der SAP Business One Erweiterungen von Interesse.

In Boehmes Softwarequalitätsmodell wird zwischen der sogenannten $as - is - utility$ und der *maintainability* unterschieden. Besonders Aspekte der Maintainability, also der Wartbarkeit, wurden in den vorangehenden Kapiteln als relevant festgestellt und beschreiben in diesem Modell wie einfach ein Programm zu verstehen oder zu ändern ist.

Modelle wie zum Beispiel das Capability Maturity Model (kurz CMM) wurden auch in Betracht gezogen, allerdings würde aufgrund der Struktur dieser Ausarbeitung nur ein geringer Teil des Modells zur Anwendung kommen können. Beispielsweise wird das CMM in fünf Bereiche geteilt, nach denen Qualität erarbeitet und bewertet wird. In dieser Ausarbeitung würde nur der erste Teil, die Initialphase relevant werden. [22]

Die gewählten Modelle und ihre jeweils beschriebenen Merkmale sind der unten aufgeführten Tabelle 6.1 auf Seite 24 zu entnehmen. Erläuterungen zu den jeweiligen Merkmalen sind unter A.3 auf Seite 72 im Anhang zu finden.

ISO 9126	**Boehme Original**	**Boehme übersetzt**
Änderbarkeit	Flexibility	Flexibilität
Effizienz	Efficiency	Effizienz
Übertragbarkeit	Portability	Übertragbarkeit
Zuverlässigkeit	Reliability	Zuverlässigkeit
Funktionalität		
Benutzbarkeit	Usability	Benutzbarkeit
	Understandability	Verständlichkeit
	Testability	Testbarkeit

Tabelle 6.1: Qualitätsmodelle mit Merkmalen

Die Merkmale Übertragbarkeit, Änderbarkeit und Benutzbarkeit aus dem ISO 9126, sowie Understandability, Usability und Flexibility aus Boehmes Qualitätsmodell sind als besonders relevant zu erachten, da diese Punkte in der Auswertung der Umfrageergebnisse aufgefallen sind.

6.2 Metriken

Um die in Kapitel 6.1 auf Seite 23 festgelegten Merkmale im Quellcode der Projekte untersuchen zu können, muss überlegt werden, welche Eigenschaften aus dem Quellcode der SAP B1 Erweiterungen der Abteilung Aussagekraft besitzen und wie diese auf die Merkmale zurückgeführt werden können.

In diesem Fall bedeutet dies, dass Software-Metriken erfasst werden, um als Betrachter eine Darstellung der Code- und Prozessqualität der Projekte des Betriebes zu erhalten, die weniger abstrakt und somit leichter einzuordnen ist. Wichtig zu beachten ist, dass Metriken keinesfalls als Ersatz oder äquivalent zu den Software-Qualitätsmerkmalen aus dem vorangegangen Kapitel 6.1 auf Seite 23 sind, sondern lediglich dazu dienen, bestimmte Eigenschaften eines Programms aufzuzeigen. Welche Schlüsse und Folgen daraus gezogen werden hängt zum einen vom Projekt selber, zum anderen von den festgelegten Zielen und Merkmalen ab.

Metriken, die mit wenig Aufwand zu erfassen und auszuwerten sind, sind beispielsweise die Lines of Code[1]. Diese können leicht je Projekt analysiert und verglichen werden, sind jedoch ohne Kontext und Einordnung wenig aussagekräftig. In Bezug auf die Dokumentation eines Projektes können die Comment Lines of Code[2] genutzt werden, um festzustellen, welcher Anteil des Quellcodes kommentiert ist. Weiterhin sollten Stilmetriken[3], wie beispielsweise Namenskonventionen oder Programmstrukturen betrachtet werden.

Da die Wartbarkeit des Quellcodes laut der Umfrage in Kapitel 4 auf Seite 13 als besonders relevant erachtet wird, sollten Metriken betrachtet werden, die Auskunft über diese geben können. Die Anzahl von Bugs, Code Smells und der technischen Schuld innerhalb des Quellcodes können eine gute Einschätzung dazu liefern und lassen sich mithilfe verschiedener Werkzeuge[4] aus dem Quellcode extrahieren.

Auch Metriken, die die Komplexität und Effizienz eines Programms beschreiben, können aussagekräftig sein. So sind beispielsweise die McCabe-Metrik [12] oder die Testabdeckung Metriken, die sich gut dazu eignen, Aspekte, wie die Komplexität eines Programms, zu quantifizieren. Diese werden allerdings aufgrund der Fokussierung auf Metriken zur Lesbarkeit und Darstellung in dieser Ausarbeitung weniger stark betrachtet.

6.3 Werkzeuge

Bestimmte Metriken lassen sich mit dem bloßen Auge oder durch manuelles Betrachten des Quellcodes häufig nicht ohne einen erheblichen Zeitaufwand erfassen. Werkzeuge sollen im Kontext dieser Ausarbeitung dabei helfen, Stilmetriken leichter zu erfassen, zu analysieren und Bugs, Code Smells und andere Schwachstellen in den Projekten entdecken und graphisch aufzubereiten.

Ein Werkzeug, welches sich für einen Großteil der Betrachtung anbietet, ist SonarQube, ein Programm zur Analyse von Quellcode. SonarQube ist eine open-source Anwendung unter der GNU Lesser General Public License und kann sowohl in einer kostenlosen, als auch einer kostenpflichtigen Enterprise-Version genutzt werden. SonarQube wurde im Rahmen dieser Ausarbeitung für eine statische Analyse der untersuchten Quellcodes eingesetzt. [24] SonarQube wurde für die Nutzung in C#-Projekten konfiguriert, um die Analyse auf den offiziellen Richtlinien von Microsoft für die Entwicklung von C#-Projekten basieren zu lassen. Auf Basis der Ergebnisse der SonarQube-Analysen wurden Schlüsse zu der Softwarequalität der betrachteten Projekte gezogen.

[1] Abgekürzt LOC, geben die gesamten Zeilen eines Programms oder einer Klasse an. Wenige LOC pro Klasse können Hinweise auf eine gute Programmstruktur geben.
[2] Abgekürzt CLOC, geben an, wie viele Zeilen eines Programms Kommentare enthalten.
[3] Eher subjektive, darstellerische Eigenschaften des Codes
[4] Siehe auch Kapitel 6.3 auf Seite 25

Um mögliche Stilmetriken zu erfassen, wird das Checkstyle-Tool StyleCop 6.1 [4], eine Erweiterung für Visual Studio, eingesetzt. Es kann zur statischen Analyse von Quellcode genutzt werden und wird in dieser Ausarbeitung benutzt, um Schwachstellen in der Darstellung des Quellcodes der einzelnen Projekte aufzuzeigen.

Die Entwicklungsumgebung, in der alle Projekte entwickelt, getestet und im Rahmen dieser Ausarbeitung analysiert werden, ist Visual Studio 2017, welches seit mehreren Jahren für die Entwicklung von allen SAP B1 Erweiterungen innerhalb der Abteilung genutzt wird. Besonders für die Implementation[5] wird Visual Studio 2017 genutzt, um die, in den Richtlinien vorgeschlagenen, Verbesserungen umzusetzen.

6.4 Analyse der Projekte

Um die festgelegten Qualitätsmerkmale aus den betrachteten Modellen konkret zu überprüfen und einzuordnen, werden diese als Grundlage für eine Analyse von Praxisbeispielen genutzt. Im Rahmen dieser Ausarbeitung werden drei, von der Abteilung entwickelte, SAP B1 Erweiterungen betrachtet und auf die folgenden Gesichtspunkte hin analysiert:

- Softwaretests
- Wartbarkeit
- Darstellung
- Dokumentation
- Fehlerbehandlung

Dabei werden in jedem Gesichtspunkt die erkannten Schwachstellen nach Relevanz aufgeführt.

6.4.1 Softwaretests

Der Zustand der Entwicklung in Bezug auf das Testen von Software sieht zum Zeitpunkt dieser Ausarbeitung innerhalb der Abteilung folgendermaßen aus:

Kleinere Anpassungen und schnelle Einschübe werden häufig nicht getestet. Es wird lediglich überprüft, ob der Quellcode kompilierbar und ausführbar ist, um anschließend möglichst schnell ausgeliefert werden zu können. Größere Erweiterungen und Projekte werden nur manuell getestet, dabei kommen, für das jeweilige Projekt, passend konfigurierte Demoumgebungen, zum Einsatz. Diese simulieren die Systeme der Kunden. Manuelles Testen wird dazu genutzt, herauszufinden, wie genau der Ablauf einer Erweiterung aus Benutzersicht in SAP B1 abläuft und ob dabei grobe Fehler oder Bugs

[5] siehe dazu Kapitel 8 auf Seite 47

besonders auffallen. Aus diesen Umständen können Situationen entstehen, in denen der Kunde mit Inbetriebnahme einer Erweiterung ein nicht komplett getestetes Produkt erhält.

Die starke Bindung von SAP B1 Erweiterungen an SAP B1 Instanzen erschwert es, automatisiert und flächendeckend zu testen. Hinzu kommt, dass für das Testen von SAP B1 Erweiterungen nur wenige offizielle Ressourcen zur Verfügung stehen und die Entwicklung von eigenen Testmethoden innerhalb der Abteilung bisher zeitlich nicht umgesetzt werden konnte. Für die Größe der typischen Projekte lohnt es sich innerhalb der Abteilung häufig nicht, Systeme für automatisiertes Testen zu entwickeln. Diese wären entweder nur unter enormem Mehraufwand möglich[6], oder aber wenig aussagekräftig[7].

Der Einsatz von externen Programmen, wie beispielsweise dem SAP Business One Test Composer [26], erleichtert zwar das Testen von SAP B1 Erweiterungen durch das automatisierte Erstellen von Testdaten, kann allerdings keine Tests der Software ersetzen.

Testabdeckung des Quellcodes und weitere Aspekte von automatisierten Softwaretests werden in dieser Analyse im Rahmen der Ausarbeitung aufgrund von zu hohem Aufwand nicht betrachtet. Der Fokus dieser ersten Iteration soll auf unmittelbaren Schwachstellen des Quellcodes liegen und als grundlegende Basis für eine langfristige Qualitätssicherung und Verbesserung dienen. In Kapitel 9.2 auf Seite 64 wird darauf noch einmal eingegangen und erörtert, wann Softwaretests in die Arbeit der Abteilung eingeführt werden sollten.

6.4.2 Wartbarkeit

Unter der Wartbarkeit werden alle Schwachstellen der Projekte zusammengefasst, die Verständlichkeit der Projekte verschlechtern und Änderungen in den Projekten erschweren.

In den Abbildungen A.2, A.3 und A.4 [8] sind die Ergebnisse der SonarQube-Analyse der drei Projekte zusammenfassend dargestellt. Nicht alle von SonarQube entdeckten Code-Smells sind für die Arbeit in der Abteilung und für diese Ausarbeitung relevant. Aufgrund der Verwendung des Coresuite Frameworks beispielsweise, müssen einige Konventionen verletzt werden, um einen korrekten Programmablauf in den Erweiterungen zu gewährleisten.

[6] Komponententests benötigen beispielsweise laufende Demosysteme und passend konfigurierte Simulationen innerhalb der Testklassen
[7] Unit-Tests für ausgewählte Methoden auf grundlegende Eigenschaften
[8] Zu finden im Anhang unter A.4 auf Seite 73

Schlecht wartbare Programmierung

Besonders auffällig sind die in den Projekten auftretenden, unintuitiven und nicht-optimalen Programmierungen. Diese haben keinen direkten Einfluss auf korrekte Programmabläufe, erschweren jedoch das Verständnis von einzelnen Zeilen oder ganzen Blöcken im Quellcode.

Beispielsweise sind in Vergleichen häufig invertierte Ausdrücke zu finden, diese entstanden höchstwahrscheinlich durch schnelle Einschübe in den Quellcode und eine fehlende Überprüfung des Geschriebenen. Die Quellcodes 6.1 auf Seite 28 und 6.2 auf Seite 28 zeigen dies anhand von Beispielen aus den Projekten.

```
if (!(activeRight < 0)
&& !(activeRight >= rightsGrid.Rows.Count - 1)
&& !(activeUser.Equals("")))
```

Quellcode 6.1: Beispiel für invertierte Ausdrücke

```
if (!(g.DataTable.GetValue(1, i).Equals(null))
&& !(g.DataTable.GetValue(1, i).Equals("-1"))
&& !(g.DataTable.GetValue(1, i).Equals(""))
&& !(g.DataTable.GetValue(0, i) == null)
&& !(alreadyExists(tmp = new Beleg(g.DataTable.GetValue(1, i),
    ↪ g.DataTable.GetValue(0, i), frm))))
```

Quellcode 6.2: Beispiel für nicht-optimale Ausdrücke

Hier sollten die üblichen Konventionen betrachtet werden, SonarQube beispielsweise schlägt an dieser Stelle vor, die in Quellcode 6.3 auf Seite 28 vorgeschlagenen Änderungen vorzunehmen, um sowohl Klammern, als auch invertierte Ausdrücke zu entfernen und somit die Lesbarkeit dieser Vergleiche zu erhöhen. Diese Änderungen entstehen durch eine korrekte Anwendung der von Microsoft vorgeschlagenen Vergleichsoperatoren für die Nutzung in C#. [21] Der Quellcode 6.3 auf Seite 28 zeigt eine mögliche Verbesserung eines solchen Ausdrucks.

```
if (g.DataTable.GetValue(1, i) != null
&& g.DataTable.GetValue(1, i) != "-1"
&& g.DataTable.GetValue(1, i) != ""
&& g.DataTable.GetValue(0, i) != null
&& !alreadyExists(tmp = new Beleg(g.DataTable.GetValue(1, i),
    ↪ g.DataTable.GetValue(0, i), frm)))
```

Quellcode 6.3: Invertierte Ausdrücke korrigiert

Weitere Schwachstellen, die den Quellcode der Projekte schwerer verständlich machen, sind nicht benötigte Umwandlungen von Datentypen. Diese sorgen nicht nur dafür, dass der Quellcode insgesamt länger und komplexer wird, sondern können auch die

Performance des Programms verschlechtern. Bei einem *cast*[9], wird das Ergebnis dieser Umwandlung in einer lokalen Variable gespeichert und als Ergebnis weiterverwendet. Bei häufigen Aufrufen kann dies dazu führen, dass viele Variablen ohne einen Nutzen gespeichert werden. [13] Quellcode 6.4 auf Seite 29 zeigt dies an einem Beispiel.

```
int activeOrder =
    ↪ (int)Convert.ToInt32(rightsGrid.DataTable.GetValue("U_Order",
    ↪ activeRight));
int previousOrder =
    ↪ (int)Convert.ToInt32(rightsGrid.DataTable.GetValue("U_Order",
    ↪ activeRight + 1));
```

Quellcode 6.4: Beispiel für nicht benötigte Umwandlung von Datentypen

Eine dritte Auffälligkeit ist in der Nutzung von Klammern in Ausdrücken zu finden. Der Übersichtlichkeit halber sollten Klammern nur dort eingesetzt werden, wo es absolut notwendig ist. Innerhalb des Quellcodes der Projekte sind mehrere Stellen zu finden, an denen Ausdrücke ein- oder sogar mehrfach mit Klammern versehen sind. Im Quellcode 6.5 auf Seite 29 ist dies zu sehen.

```
userGrid = ((SAPbouiCOM.Grid)(this.GetItem("UserGrid").Specific));
```

Quellcode 6.5: Beispiel für übermäßige Nutzung von Klammern

Wie auch die anderen betrachteten Schwachstellen, schränkt dies die Lesbarkeit und Einfachheit des Quellcodes ein, der Leser muss versuchen die Klammerpaare einander zuzuordnen, um zu verstehen, welche Teile diese abgrenzen sollen.

Enge Kopplung und niedrige Kohäsion

Alle betrachteten Projekte zeichnen sich durch eine hohe Anzahl von globalen Variablen und Methoden aus. Dies ist besonders bei UI-Elementen und Methoden für die Verbindung zu SAP B1 Instanzen nicht vermeidbar. In diesem Zuge werden häufig auch weitere Komponenten nach dem gleichen Schema angelegt.

Dies sorgt für eine enge Kopplung innerhalb der Erweiterungen, das bedeutet, dass Komponenten abhängig voneinander sind und nur schwer voneinander getrennt werden können, ohne den Programmablauf zu beeinflussen. Besonders in der Wartbarkeit eines Programms hat dies Auswirkungen. Mit jeder Änderung an einer Komponente muss sichergestellt werden, dass alle davon abhängigen Komponenten noch korrekt funktionieren, was bei einer engeren Kopplung zu höherem Aufwand führt.

Eine enge Kopplung bedingt im Umkehrschluss häufig auch eine niedrige Kohäsion. Hohe Kohäsion bedeutet, dass eine Komponente für genau eine logische Einheit oder

[9] Explizite Umwandlung in einen Datentyp, beispielsweise von Integer zu String

Aufgabe zuständig ist. Änderungen oder Anpassungen an der Funktionsweise einer Erweiterung können mit einer niedrigen Kohäsion nur mit größerem Aufwand umgesetzt werden, da Änderungen der Logik des Programms nicht nur an genau einer Stelle im Quellcode Auswirkungen haben können.

Konkret betrachtet sind in den analysierten Projekten eine Vielzahl von globalen Klassenvariablen aufgefallen, die je nach Aufgabe in Methoden der Klasse verschoben werden könnten. Damit könnte eine losere Kopplung erreicht werden. Weiterhin sollten die Klassen stärker nach ihrer logischen Aufgabe geteilt werden, um eine höhere Kohäsion zu erzielen.

Eine grundlegende Überarbeitung der angewandten Architektur innerhalb der Erweiterungen könnte in Zukunft genutzt werden, um diese Aspekte weiter zu verbessern.

SQL-Abfragen

Häufig kommt es vor, dass eine SAP B1 Erweiterung Daten aus einer der SAP B1 Datenbanktabellen lesen, ändern oder schreiben muss.

Diese SQL-Abfragen werden in den Projekten innerhalb des Quellcodes mithilfe von String-Konkatenation erzeugt, was zum einen die Performance des Programms verringern kann, zum anderen können gerade sehr lange Abfragen schnell unübersichtlich werden. Eine Auslagerung von langen Abfragen in SQL-Dateien oder die Erstellung mithilfe eines Stringbuilders in einer eigenen Klasse sind als Alternativen denkbar.

Auffällig ist, dass keine Prüfungen der auszuführenden SQL-Abfragen stattfinden. So ist es beispielsweise möglich, per SQL-Injection Schäden an der Datenbank anzurichten. Eine Nutzung des C#-Befehls **Prepare** [19] zur Vorbereitung und korrekten Überprüfung der Parameter der SQL-Abfrage wird allerdings sowohl vom SAPBusinessOneSDK, als auch dem Coresuite Framework nicht unterstützt. Potentiell könnten mit der Anwendung von Stringbuildern die Parameter manuell überprüft und ausgetauscht werden, um eine mögliche SQL-Injection zu verhindern. Darauf wird allerdings aufgrund von zeitlichen Begrenzungen bei der Erweiterungs-Entwicklung verzichtet. SAP B1 Erweiterungen werden zudem nur von den jeweiligen Kunden benutzt und nur ein minimaler Teil der möglichen Eingaben kann kritischen Schaden anrichten, was den zusätzlichen Aufwand häufig nicht rechtfertigt.

Access Modifier

In den betrachteten Projekten fehlen stellenweise die Access Modifier für Klassen, Methoden oder Variablen und sind über die verschiedenen Klassen hinweg inkonsistent benutzt.

Es kann vor allem bei fehlenden Access Modifiern zu Verwirrung kommen, da diese, abhängig von dem darauf folgenden Objekt, eine andere Bedeutung haben. Objekte einer Klasse werden beispielsweise ohne explizite Access Modifier als *private* bezeichnet. [27] Fehlt einem Leser dieses Wissen, so kann es zu Verwirrungen kommen, die durch eine durchgängige Anwendung von Zugriffsbeschränkungen verhindert werden können.

Zusätzlich stellt der Einsatz von *private* sicher, dass auf Objekte nicht von außen zugegriffen werden kann. Dies führt zu einer loseren Kopplung, da keine unbemerkten Abhängigkeiten im Quellcode entstehen.

Es sollte darauf geachtet werden, passende Access Modifier zu nutzen, um die Zugriffe auf Objekte gemäß ihrer Nutzung einzuschränken und dabei auf eine einheitliche Anwendung dieser zu achten.

Ungenutzter Quellcode

Ungenutzte Variablen oder leere Methoden erfüllen innerhalb des Quellcodes keinen Zweck und sollten entfernt werden. Gerade definierte und initialisierte Variablen können leicht zu Verwirrung führen, wenn nicht deutlich erkennbar ist, dass diese im Quellcode nicht weiter genutzt werden. Die eingesetzte Entwicklungsumgebung[10] markiert diese zwar, wie in Abbildung 6.1 auf Seite 31 zu sehen ist, dies ist allerdings leicht zu übersehen und kann den Quellcode bei häufigem Auftreten unübersichtlicher machen.

```
int UnusedNumber = 1;
```

Abbildung 6.1: Von Visual Studio markierte, ungenutzte Variable

Viele Code Smells entstehen durch auskommentierten, alten Code, häufig Überbleibsel aus älteren Versionen der Erweiterung, die während der Entwicklung angelegt und später verworfen wurden. Dies erschwert die Lesbarkeit des Codes und sorgt effektiv dafür, dass die betroffenen Klassen und Methoden länger werden, ohne einen konkreten Nutzen für das Programm zu liefern.

Im Quellcode A.3 auf Seite 74 im Anhang ist ein besonders auffälliges Beispiel dafür zu sehen, an dieser Stelle wurde der gesamte Inhalt einer Methode auskommentiert, was schlussendlich dafür sorgt, dass sie keinen Nutzen mehr erfüllt und entfernt werden könnte.

[10] Visual Studio 2017

Zusammenfassung zur Wartbarkeit

Die Schwachstellen in der Wart- und Lesbarkeit aller betrachteten Projekte werden vor allem durch unsaubere Programmierung hervorgerufen. Leere Methoden, ungenutzte Variablen und auskommentierte Codeblöcke füllen den Quellcode mit irrelevanten Inhalten, die im besten Fall für eine schlechtere Übersicht, im schlimmsten Fall für Fehler sorgen können. An vielen Stellen wirkt der Quellcode unfertig oder verwirrend. Dies wird häufig durch kurze und schnelle Einschübe in der Korrektur von Fehlern hervorgerufen.

Diese Aspekte machen es besonders für Entwickler, die nicht unmittelbar am Projekt beteiligt sind, schwerer, in den Code einzusteigen oder diesen zu verstehen. Zusammenfassend kann gesagt werden, dass die Wartbarkeit der Erweiterungen durch die betrachteten Schwachstellen eingeschränkt und erschwert wird und somit zu einer Verminderung der Softwarequalität für dieses Projekt führt. Die Richtlinien sollen auf die hier angesprochenen Schwachstellen und deren Auswirkungen auf die Arbeit mit den Projekten eingehen, um dafür zu sorgen, in zukünftigen Projekten ähnliche Probleme zu vermeiden.

6.4.3 Darstellung

Unter der Darstellung sind Schwachstellen zusammengefasst, die keine Auswirkungen auf die Funktionsweisen der betrachteten Projekte haben, sondern lediglich stilistischer Natur sind. Doch auch diese Schwachstellen können es erschweren, den Quellcode der Projekte zu lesen und zu verstehen, was schlussendlich zu Fehlern oder schlechterer Code-Qualität führen kann.

Namensschemata

In den betrachteten Erweiterungen fällt auf, dass Namen vom Standard-C#-Schema abweichen und verschiedene, andere Notationen nutzen. Es werden für die Groß- und Kleinschreibung beispielsweise Einsätze von Java-Notationen[11] oder die Verwendung von snake_case[12] festgestellt.

Innerhalb eines jeweiligen Projektes hat dies keine unmittelbaren Auswirkungen, solange es einheitlich bleibt, allerdings kann es schnell zu Verwirrungen kommen, wenn andere Projekte ein anderes Namensschema verwenden. So verletzen beispielsweise die Klassennamen eines der betrachteten Projekte, zu sehen in Abbildung 6.2 auf Seite 33, die Namenskonventionen für Klassen in C# durch den Einsatz von camelCase. [17]

[11] Methoden- und Variablennamen werden klein geschrieben
[12] Alles wird klein geschrieben und mit Unterstrichen verknüpft

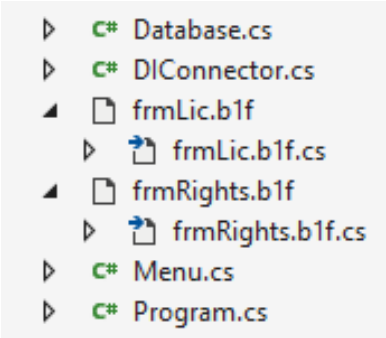

Abbildung 6.2: Beispiel für Klassennamen, die Konventionen zur Groß- und Kleinschreibung verletzen

Der Quellcode 6.6 auf Seite 33 zeigt beispielhaft Namen von Variablen, die nicht den offiziellen Vorgaben von C# [18] entsprechen. Diese sollten alle nach PascalCase und ohne Unterstriche benannt sein, um dem, von Microsoft vorgeschlagenen, Standard zu entsprechen.

```
private SAPbouiCOM.Button button_ok;
private SAPbouiCOM.Button button_ca;
private StaticText maxLicLabel;
private StaticText dateLabel;
```

Quellcode 6.6: Beispiel für Variablennamen, die C# Namenskonventionen verletzen

Weiterhin betrachtet wird die Art der Benennung von Bezeichnern. Um eine möglichst hohe Übertragbarkeit und intuitive Benennung von Bezeichnern zu erhalten, sollten alle Namen in englisch vergeben werden. Dies hat nicht nur den Vorteil, dass international keine Kommunikationsprobleme auftreten können, sondern erleichtert auch das Verketten von Schlüsselwörtern und vermeidet Umlaute.

In den Projekten sind zum größten Teil englische Bezeichner zu finden, doch an einigen wenigen Stellen wurden deutsche Bezeichner verwendet. In Quellcode 6.7 auf Seite 34 ist ein Beispiel dafür zu sehen, dies sticht aus dem Quellcode heraus und sorgt für eine uneinheitliche Benennung, die schwieriger zu lesen ist, da der Leser zwischen zwei Sprachen wechseln muss.

Im Quellcode 6.7 auf Seite 34 ist außerdem eine weitere Schwachstelle in der Namensgebung zu sehen. Bezeichner wie frm oder g nähern sich der ungarischen Notation [20] an, einer Art der Namensgebung, die dafür kritisiert wird, unintuitiven und schwer lesbaren Code zu produzieren. [28]

Um möglichst intuitive Namen zu erhalten, sollten alle Elemente nach ihrer Funktion benannt werden und aussagekräftige, für sich sprechende[13] Namen erhalten. [23] Auch für diese Fälle gilt, dass Einheitlichkeit und Konsistenz im Normalfall an erster Stelle stehen sollten.

[13] Quellcode, der sich durch seine Struktur und Benennung selbst erklärt

```
private Form frm, callingFrm;
private Grid g, backgroundGrid;
List<Beleg> activeList;

// Konstruktor
public Verknuepfungsplan(string frmID)
{
        this.callingFrm = Form.GetFormFromUID(frmID);
        CreateForm();
        createRelationsTable();
        showVerknuepfungsPlan();
}
```

Quellcode 6.7: Beispiel für deutsche Bezeichner

```
private StaticText StaticText5;
private StaticText StaticText6;
private StaticText StaticText7;
```

Quellcode 6.8: Beispiel für wenig aussagekräftige Bezeichner

Zudem sollten Namen, wie in Quellcode 6.8 auf Seite 34 zu sehen sind, vermieden werden. Diese geben keine Auskunft über die Verwendung und die Funktion. Leser, besonders neue Entwickler, müssen erst durch den Kontext im restlichen Quellcode herausfinden, wofür diese Variablen genutzt werden.

Struktur von Klassen

Weitere Schwachstellen sind in der Struktur der Klassen in den betrachteten Projekten zu finden. Konstruktoren oder die Definitionen von Variablen finden teilweise am Ende der Klasse oder zwischen Methoden statt. Es ist nicht immer klar erkennbar, wo bestimmte Elemente einer Klasse innerhalb des Quellcodes zu finden sind. Die Abbildung 6.3 auf Seite 35 zeigt, wie Klassen laut den von Stylecop 6.1 vorgegebenen Richtlinien für die Strukturierung von C#-Klassen [30], aussehen könnten.

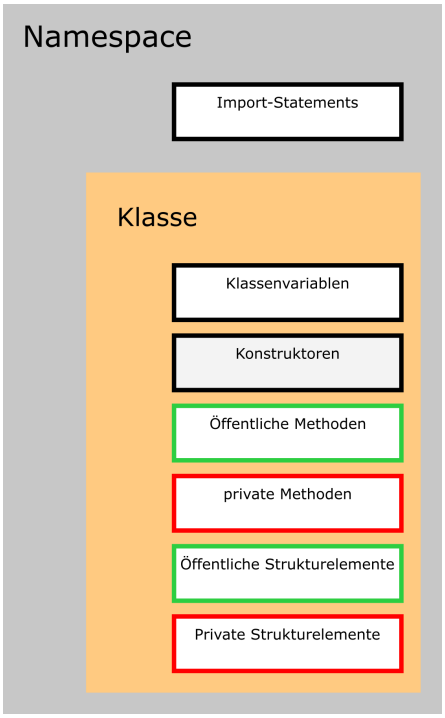

Abbildung 6.3: Vorgeschlagene Struktur von Klassen

Um es zu erleichtern, auf den ersten Blick zu erkennen, welcher Klasse ein Code-Element zugehörig ist, sollten Methodenaufrufe und Variablen mit *this.* versehen werden. Dadurch wird eindeutig festgelegt, dass ein Aufruf für Elemente aus der aufrufenden Klasse gilt, dies sorgt allerdings für eine leichte Verlängerung der einzelnen Aufrufe. Ein sprechender und gut dokumentierter Quellcode sollte die absolute Notwendigkeit für diese eindeutige Bezeichnung überflüssig machen, kann allerdings gerade bei fehlender Dokumentation oder verwirrenden Programmstrukturen eine Verbesserung der Lesbarkeit bieten.

Import-Statements

Import Statements sind in der Entwicklung von SAP B1 Erweiterungen unverzichtbar. In beinahe jeder Klasse der betrachteten Projekte finden sich Objekte aus den Bibliotheken *SAPbouiCOM* oder *SAPbobsCOM* wieder. Die Aufrufe dieser Objekte können sich teilweise sehr lang gestalten und den Quellcode so schwerer lesbar und weniger leicht nachvollziehbar gestalten, wenn kein Vorwissen über die importieren Objekte und Funktionen besteht. In den Quellcodes A.1 und A.2 auf Seite 73 im Anhang ist dies beispielhaft dargestellt.

Eine potentielle Lösung dafür wäre, eine einheitliche Struktur zu entwickeln, nach der Import-Statements am Anfang einer jeweiligen Klasse gesetzt werden, um den eigentlichen Quellcode übersichtlicher zu gestalten. Die Nutzung eines Alias' beispielsweise, kann längere Import-Statements verkürzen und die Anwendung im eigentlichen Quellcode einer Klasse übersichtlicher machen.

Spacing

Fehlende Leerzeilen nach geschlossenen, geschweiften Klammern, sowie zu viele überflüssige Leerzeilen, die allgemein im Quellcode zu finden sind, wurden durch die Analyse der Quellcodes aufgezeigt.

Auch die Verwendung von Leerzeichen und Tabulatoren ist in den betrachteten Erweiterungen aufgefallen, so sind diese nicht einheitlich über alle Projekte hinweg konfiguriert. Eines der Projekte wurde mit acht Leerzeichen je Tabulator entwickelt, während die anderen mit 4 Leerzeichen je Tabulator geschrieben wurden. Solche Unterschiede können dafür sorgen, dass ein Projekt nicht korrekt in einer anderen Entwicklungsumgebung dargestellt werden kann, oder aber Probleme bei der Versionskontrolle entstehen können. Dies führt zu einem Mehraufwand für die zuständigen Entwickler.

Die Auswirkungen davon auf die Lesbarkeit des Quellcodes sind zwar minimal, allerdings sind die verursachenden Zeilen leicht zu beheben und können angepasst werden, um die einzelnen Klassen übersichtlicher zu machen.

Zusammenfassung Darstellung

Die darstellerischen Schwachstellen erschweren es vor allem, die Quelltexte einfach zu lesen und sorgen dafür, dass Zeit dafür genutzt werden muss, die Eigenheiten des Quellcodes zu verstehen, bevor deutlich wird, worum es inhaltlich geht. Gerade die oben genannten Aspekte können davon profitieren, vereinheitlicht und durch die Richtlinien vorgegeben zu werden, um so einheitliche Schemata für die Erstellung von Quellcode zu erhalten.

Neue Entwickler können sich somit auf einheitliche Strukturen und Vorgaben verlassen und somit leichter in ein Projekt einarbeiten, da sichergestellt ist, dass die Darstellung in allen Projekten das gleiche bedeutet.

6.4.4 Dokumentation

Unter dem Punkt Dokumentation werden Schwachstellen in den Kommentaren und der allgemeinen Dokumentation innerhalb des Quellcodes zusammengefasst.

Kommentare

Durchschnittlich ist in den Projekten ein Anteil von circa 15% - 20% der Zeilen mit einem Kommentar versehen. Oberflächlich betrachtet kann davon ausgegangen werden, dass alle relevanten Zeilen beschrieben und erklärt werden. Bei genauerem Betrachten wird allerdings deutlich, dass viele dieser Kommentare entweder alten, ungenutzten Code

enthalten oder Inhalte nur unzureichend beschreiben. An anderen Stellen enthalten sie zu viele Informationen oder sind überflüssig.

Kommentare sollten erklären, warum eine Komponente im Quellcode existiert, statt zu erklären, wie diese arbeitet. Damit können Elemente des Quellcodes in einen Kontext der Anwendung gestellt werden, während sprechende Namen und Strukturen die Erklärung der Funktionsweise übernehmen können.

Es muss überlegt werden, an welchen Stellen im Quellcode Dokumentation wirklich nötig ist. Die Dokumentation von Quellcode, der sich selbst erklärt[14], ist beispielsweise nicht sinnvoll und sollte vermieden werden. Das Beispiel in Quellcode 6.9 auf Seite 37 zeigt ein häufig auftretendes Beispiel aus den betrachteten Projekten.

```
// Button erstellen - OK Funktionen
Button okButton = Button.CreateNew("ok_btn");
okButton.Value = "OK";

// Laden der Form
frm.Load();
```

Quellcode 6.9: Beispiel von überflüssigen Kommentaren

Es gilt abzuwägen wo und in welchem Umfang Kommentare und Dokumentation innerhalb des Quellcodes sinnvoll sind, fest steht jedoch, dass qualitativ hochwertige und aussagekräftige Kommentare das Einarbeiten in ein Projekt beschleunigen. Auch Änderungen an den Erweiterungen können durch eine gute Dokumentation leichter durchgeführt werden.

Dokumentation

In den betrachteten Projekten fehlt eine ausführliche Dokumentation der Klassen und Methoden durch beispielsweise Klassen- oder Methodenköpfe. Dies sorgt dafür, dass notwendige Kontextabgrenzungen und Informationen über die Funktionen der Klassen oder Methoden verloren gehen und häufig nur unzureichend durch Line-Kommentare beschrieben werden.

Methoden sind mit einzeiligen Kommentaren ausgestattet, die beschreiben, was eine jeweilige Methode macht, doch Informationen zu dem Zweck der Methode, den darin genutzten Parametern und Rückgabewerten und in welchem Kontext sie zu verstehen ist, fehlen im gesamten Quellcode.

Dokumentationsköpfe sollten als Ergänzung zu einem sprechenden Quellcode dienen und das Verständnis sowie die Lesbarkeit vereinfachen, besonders für Entwickler, die versuchen sich in ein Projekt einzuarbeiten. Weiterhin sind an vielen Stellen Kommentare vorhanden, die beschreiben, was mit einer jeweiligen Variable oder Codezeile

[14] Also sprechend ist

gemeint ist. Solche Kommentare sollten im Optimalfall durch eine aussagekräftige und sprechende Benennung der betreffenden Zeilen ersetzt werden. [23]

Änderungsprotokolle

Änderungen am Quellcode werden in den betrachteten Projekten durch Line-Kommentare an den veränderten Zeilen festgehalten. Im Quellcode 6.10 auf Seite 38 ist dies zu erkennen.

```
if ( CardCode != CardCodeNext){
// MT 30.07.2018 - geht auch nach Invoice.Lines.Add();
MakeInvoiceHead(ref Invoice, CardCode, Datum);
AddInvoice(Invoice, QueryStr);} // MT 30.07.2018
```

Quellcode 6.10: Beispiel von Änderungsprotokollen als Kommentare im Code

Eine solche Art der Dokumentation hat den Nachteil, dass keine vollständige Auflistung der Änderungen möglich ist, ohne vorher den gesamten Code nach einzelnen Änderungen anhand dieser Kommentare zu durchsuchen. Es ist sinnvoller, die Änderungen als Protokoll an einer einheitlichen Stelle im Quellcode festzuhalten, um es für Entwickler leichter zu machen, diese zu finden.

In den Dokumentationsköpfen der Klassen beispielsweise, können solche Änderungsprotokolle eingefügt werden. Alternativ sollte durch die Versionskontrolle sichergestellt werden, dass eine aussagekräftige Versionierung der einzelnen Projekte genutzt werden kann.

Line-Kommentare

In den betrachteten Projekten sind ausschließlich Line-Kommentare zu finden. Die beiden dabei aufgefallenen Schwachstellen waren zum einen die übermäßige Nutzung von Kommentaren, wie beispielsweise in Quellcode 6.11 auf Seite 38 zu sehen ist.

```
case "23": // Angebot
return "Angebot";
case "17": // Kundenauftrag
return "Kundenauftrag";
```

Quellcode 6.11: Beispiel für überflüssige Namensgebung durch Kommentare

Zum anderen fehlen an anderen Stellen Kommentare, wo sie vielleicht nützlich, oder sogar notwendig wären. In Quellcode 6.12 auf Seite 39 ist ein Beispiel für eine Codezeile zu sehen, die von einer kurzen Erklärung profitieren könnte. Eine Erklärung, was genau durch den Aufruf von *delegate* geschieht, könnte beispielsweise dazu beitragen, diese Codezeile besser zu verstehen.

```
menuFaktura.AddHandler_Click(delegate
    ↪ (SwissAddonFramework.UI.EventHandling.MenuEvents.MenuClick
    ↪ eventVal)
{
Modules.FakturaModul Faktura = new Modules.FakturaModul();
});
```

Quellcode 6.12: Beispiel für fehlende Line-Kommentare

Im folgenden Quellcode 6.13 auf Seite 39 ist ein weiteres Beispiel für eine nötige Beschreibung durch Kommentare zu finden. Ohne Beschreibung müsste ein Handbuch oder andere Hilfe genutzt werden, um herauszufinden, wofür beispielsweise „OQUT“ steht.

```
// VERKAUF
case "23": // Angebot
tableNames[0] = "OQUT";
tableNames[1] = "QUT1";
break;
```

Quellcode 6.13: Beispiel für nötige Namensgebung durch Kommentare

Line-Kommentare und deren Einsatz durch Richtlinien vorzugeben, stellt sich als schwierig heraus und muss durch Erfahrungen in der Entwicklung unterstützt werden. Es können nur Vorschläge gemacht werden, die Tipps und Ideen geben, wann und an welchen Stellen Line-Kommentare sinnvoll sein könnten. Festzuhalten bleibt jedoch, dass eine Dokumentation des Quellcodes durch Kommentare für die Arbeit der unterschiedlichen Entwickler nicht verzichtbar ist.

Zusammenfassung Dokumentation

Die Dokumentation der Projekte ist an vielen Stellen unzureichend und sorgt für keine eindeutige und aussagekräftige Erklärung des Quellcodes. Zudem wird kaum eines der genutzten Elemente[15] in einen Kontext innerhalb des Programms gesetzt.

In den Richtlinien werden daher Möglichkeiten für die Dokumentation von Quellcode in Form von Dokumentationsköpfen und relevanten Line-Kommentaren genannt.

6.4.5 Fehlerbehandlung

Zwar werden in dieser Ausarbeitung keine Softwaretests betrachtet[16], allerdings sollen Schwachstellen in der Fehlerbehandlung der betrachteten Projekte zusammengefasst werden.

[15] Klasse, Methode, Variable, usw.
[16] Siehe dazu Kapitel 6.4.1 auf Seite 26

```
try
{
        //  If the manu already exists this code will fail
        oMenus.AddEx(oCreationPackage);
}
catch (Exception e)
{

}
```

Quellcode 6.14: Beispiel für eine Exception ohne Fehlerbehandlung

Ausnahmebehandlung

Die meisten Schwachstellen in diesem Bereich, beziehen sich auf die Ausnahmebehandlung innerhalb des Quellcodes. Dabei werden Exceptions zwar abgefangen, allerdings findet keine weiterführende Behandlung dieser Ausnahmen statt. Der Quellcode 6.14 auf Seite 40 zeigt ein Beispiel aus einem der betrachteten Projekte für eine fehlende Ausnahmebehandlung.

Ohne eine aussagekräftige Behandlung einer Exception oder passender Kommentare, warum diese nicht notwendig ist, ist es für Entwickler schwer zu erkennen, wann genau diese Exception auftritt, wie mit dieser umgegangen werden soll und welche Relevanz sie hat. Eine Fehlerbehandlung auf Basis von Exceptions ist somit auch nicht möglich, da keine Sicherheit für die korrekte Behandlung dieser gewährleistet ist. Diese Ausnahmefälle eignen sich in den betrachteten Projekten hauptsächlich dazu, herauszufinden, an welcher Stelle im Programm eine Ausnahme auftritt, eine Weitergabe an andere Programmteile oder eine direkte Behandlung des Fehlerfalls finden nicht statt.

Es scheint, als wären die Exceptions nur abgefangen worden, um die Erweiterungen kompilierbar und ausführbar zu machen.

Parameterprüfung

In vielen Methoden innerhalb der betrachteten Erweiterungen kann es sinnvoll sein, Fehlerfälle zu verhindern, indem Parameter und Rückgabewerte direkt innerhalb des Quellcodes auf korrekte oder erlaubte Werte überprüft werden. Der Aufwand in der Entwicklung ist dafür relativ klein, kann aber potentiell später auftretende Fehler vermeiden.

In den analysierten Projekten findet keine Prüfung der genutzten Parameter statt, es wird davon ausgegangen, dass diese nur erlaubte und korrekte Werte besitzen. Aufgrund der fehlenden Tests, kann dies allerdings nicht immer sichergestellt werden, was zu potentiellen Fehlerquellen innerhalb der Erweiterungen führt.

Zusammenfassung Fehlerbehandlung

Es wurden Möglichkeiten zur Fehlerbehandlung im Quellcode aufgezeigt. Die Überprüfung von Parametern oder der korrekte Einsatz von Exceptions kann dazu beitragen, Erweiterungen mit weniger Fehlern auszuliefern.

Die Richtlinien erläutern die korrekte Anwendung dieser beiden Fälle und beschreiben, was die Vorteile davon in der Entwicklung weiterer SAP B1 Erweiterungen sind. Es muss allerdings beachtet werden, dass diese Vorschläge nur als zusätzliche Maßnahmen gesehen werden sollten. Ausführliche Tests der Software können dadurch aber nicht ersetzen werden.

7 Beschreibung der Richtlinien

In diesem Kapitel werden die Ergebnisse der Analyse aus Kapitel 6 auf Seite 23 zusammengefasst und aufbereitet. Dazu werden Schlüsse gezogen, die als Grundlage für Richtlinien, Vorschläge und Vorgaben dienen können, um die Qualität der Projekte zu verbessern.

7.1 Format

Um die Richtlinien für den Betrieb umzusetzen, sollten diese für die erste Iteration des gewählten Ansatzes[1] möglichst einfach und modellbasiert[2] dargestellt werden. Zudem muss beachtet werden, dass sich die Richtlinien auf den Quellcode der SAP B1 Erweiterungen beziehen werden und leicht darauf anzuwenden sein sollten.

Um die in dieser Arbeit angesprochenen Schwachstellen und Richtlinien zu dokumentieren und bessere Prozesse und Methoden vorzugeben, eignet sich ein User Manual (oder auch Benutzerhandbuch). Dieses kann im Laufe der Zeit und mit weiteren Iterationen, der in Kapitel 3 auf Seite 9 erarbeiteten Methodik, erweitert und mithilfe von Erfahrungswerten angepasst werden. Diese erste Iteration gibt dabei im Nutzerhandbuch Konventionen zu Quellcode, Benennung, Dokumentation und Fehlerbehandlung vor. Spätere Iterationen können sich beispielsweise mit der Optimierung von Entwicklungsprozessen, Softwarearchitektur oder Sicherheitsfragen beschäftigen und somit zu einer stetig wachsenden Sammlung von Erfahrungen vereint werden, mit dem Ziel, die Software- und Prozessqualität innerhalb der Abteilung zu verbessern.

Das User Manual wird in mehrere Bereiche mit unterschiedlichen Schwerpunkten eingeteilt. So wird zu Beginn des Dokumentes erläutert, wie die Richtlinien zu benutzen sind. In den weiteren Kapiteln werden Richtlinien zur Verbesserung der Wartbarkeit, Darstellung, Dokumentation und Testabdeckung nacheinander betrachtet. In der folgenden Tabelle 7.1 auf Seite 44 ist ein grober Aufbau des User Manuals zu finden und was in den jeweiligen Teilen erklärt und vorgegeben werden soll:

[1] siehe dazu Kapitel 3.3 auf Seite 10
[2] Als eine Art Referenzmodell

Kapitel	Inhalt
Benutzung	Wie sind diese Richtlinien zu benutzen?
Wartbarkeit	Behebung und Vermeidung von Code Smells
Darstellung	Stilistische Vorgaben zum Erstellen von Quellcode
Dokumentation	Vorgaben für Dokumentation des Quellcodes
Fehlerbehandlung	Richtlinien für Fehlerbehandlung und -Vermeidung

Tabelle 7.1: Inhalt des User Manuals

Die endgültigen und ausformulierten Richtlinien sind als User Manual im Anhang A.7 auf Seite 75 zu finden. In den folgenden Kapiteln werden die Inhalte dieses Richtlinien-Dokuments erklärt und beschrieben.

7.2 Benutzung der Richtlinien

Um einen groben Überblick über das Einsatzgebiet der Richtlinien zu erhalten und somit für jeden Leser zu verdeutlichen, wozu und wie dieses User Manual genutzt werden soll, wird vorab die Benutzung des Dokuments beschrieben. Hierbei wird der Geltungsbereich abgesteckt, dieser beinhaltet zum Stand dieser Ausarbeitung alle SAP B1 Erweiterungen aus der Entwicklung der Abteilung. Weiterhin wird erklärt, wie diese Richtlinien zu verstehen und anzuwenden sind und wozu sie nicht geeignet sind.

Die folgenden Richtlinien und Vorgaben zur Erstellung von Quellcode beziehen sich auf die Entwicklung von SAP B1 Erweiterungen. Der Fokus der Richtlinien liegt vor allem auf der grundlegenden Verbesserung der Wart- und Lesbarkeit durch Vorgaben zur Erstellung von Quellcode. Diese Richtlinien sind als Orientierung zu verstehen, wenn es darum geht, neuen Quellcode im Zuge der Entwicklung einer SAP B1 Erweiterung zu schreiben, oder aber, wenn bereits existierender Quellcode refaktoriert und verändert werden soll.

Die in diesem Dokument genannten Richtlinien sind nicht als vollständig und endgültig zu verstehen und können zu jeder Zeit, auf Basis von Feedback der verantwortlichen Mitarbeiter oder Änderungen in der Arbeitsweise der Abteilung, geändert und angepasst werden. Dadurch, dass sie im Betrieb als PDF-Dokument vorliegen werden, müssten Änderungen besprochen und anschließend von einem verantwortlichen Mitarbeiter umgesetzt werden. Sie sollen als Grundlage für eine langfristige Sicherung von Softwarequalität in der gesamten Entwicklung von SAP B1 Erweiterungen gesehen werden und stellen an dieser Stelle den ersten Schritt des gesamten Prozesses der Verbesserung der Softwarequalität dar.

7.3 Kapitel der Richtlinien

Im Folgenden werden die Kapitel der Richtlinien kurz beschrieben.

7.3.1 Wartbarkeit

In diesem Kapitel der Richtlinien werden alle Richtlinien aufgeführt, die dazu beitragen sollen, die Wartbarkeit, sowie die Übertragbarkeit und Einfachheit des Quellcodes zu verbessern. Dabei wird vor allem auf Vereinfachungen in der Programmierung eingegangen.

7.3.2 Darstellung

Unter dem Kapitel der Darstellung ist im User Manual die Sammlung von Richtlinien zu verstehen, die für die Verbesserung von stilistischen Schwachstellen zuständig sind. Dazu gehören hauptsächlich die Vereinheitlichung von Namen und Konventionen für Bezeichner, sowie Vorgaben für die genutzte Sprache, und Art der Benennung. Weiterhin werden an dieser Stelle Richtlinien für die Strukturierung von Klassen und die Verwendung von Trennzeichen oder Leerzeilen vorgeschlagen.

7.3.3 Dokumentation

In dem Kapitel Dokumentation gibt das User Manual Richtlinien vor, die dabei helfen sollen, zu verstehen, wie Inhalte im Quellcode dokumentiert werden sollten. Dabei wird auf die Dokumentation von Klassen, Methoden und Variablen eingegangen. Zusätzlich werden Vorgaben für die Erstellung von einheitlichen Bearbeitungsprotokollen und Line-Kommentaren präsentiert.

7.3.4 Fehlerbehandlung

Der letzte Teil des User Manuals befasst sich mit Ansätzen zur besseren Behandlung von Fehlern und Ausnahmefällen innerhalb des Quellcodes und gibt Richtlinien für die korrekte Benutzung von Exceptions, sowie Prüfungen von relevanten Parametern und Werten vor.

7.4 Umsetzung

Die Umsetzung der erstellten Richtlinien erfolgt in mehreren Schritten, dies ist in Tabelle 7.2 auf Seite 46 dargestellt.

Schritt	Maßnahme
1	Beispielhafte Anwendung der Richtlinien auf eine SAP B1 Erweiterung
2	Vergleich der Erweiterung vor und nach der Anwendung
3	Präsentation und Auswertung der Ergebnisse
4	Einführung der Richtlinien in die Entwicklung von neuen Projekten im Betrieb
5	Eventuelle Anwendung der Richtlinien auf bereits bestehende Projekte
6	Nutzung der Richtlinien als Grundlage für zukünftige Qualitätsverbesserungsmaßnahmen

Tabelle 7.2: Umsetzung der Richtlinien

Die ersten drei Schritte werden im Kontext dieser Ausarbeitung in den folgenden Kapiteln 8: Umsetzung der Richtlinien auf Seite 47, sowie 8.4: Auswertung der Umsetzung auf Seite 56 durchgeführt und erläutert. Dabei wird das Lagerscanner Rechteverwaltungsprojekt auf Basis der Richtlinien refaktoriert. Anschließend wird diese neue Version der Erweiterung dann mit der vorherigen Version verglichen. Die Ergebnisse daraus werden im Rahmen dieser Ausarbeitung aufbereitet und präsentiert. Die weiteren Schritte vier bis sechs werden anschließend im Betrieb vorgestellt und unter Einsatz, der in den ersten Schritten erarbeiteten, Ergebnisse unterstützt, um diese als Grundlagen für die Entwicklung von SAP B1 Erweiterungen dauerhaft einzuführen. Anschließend kann diese erste Iteration abgeschlossen und eine weitere Iteration begonnen werden, wie in Kapitel 3.3 auf Seite 10, sowie in Kapitel 9.2 auf Seite 64 beschrieben ist.

8 Umsetzung der Richtlinien

Um herauszufinden, ob die, in dieser Ausarbeitung erstellten, Richtlinien zielführend und wirksam sind, wird eines der analysierten Projekte als Veranschaulichung genutzt. Auf Basis der erstellten Richtlinien wird das Projekt modifiziert, um den darin genannten Vorgaben zu genügen. Anschließend wird es nach dieser Anpassung mit dem vorherigen Stand verglichen.

8.1 Art der Umsetzung

Die Richtlinien wurden mit besonderem Fokus auf Anwendbarkeit erstellt. In der Abteilung werden sie vorerst hauptsächlich dafür eingesetzt, neue Erweiterungen darauf aufzubauen und alte Projekte schrittweise zu verbessern. Daher ist es sinnvoll, diese Richtlinien durch eine praktische Umsetzung zu testen.

Die beispielhafte Umsetzung der Richtlinien soll anhand einer Refaktorierung erfolgen. Um eine möglichst akkurate Gegenüberstellung zu ermöglichen, wird anschließend ein „vorher-nachher“ Vergleich durchgeführt.

8.2 Refaktorierungsprojekt

Die Richtlinien werden in diesem Kapitel beispielhaft auf eine SAP B1 Erweiterung für die Rechteverwaltung eines Lagerscanners, nachfolgend Refaktorierungsprojekt[1] genannt, angewendet. Dieses Projekt zeigt trotz des relativ geringen Projektumfangs, viele der im Betrieb typischen und häufig auftretenden Schwachstellen, auf.

Die Refaktorierung der Erweiterung basiert auf der in Kapitel 6.4 auf Seite 26 betrachteten Version[2], um so eine direkte Gegenüberstellung zu ermöglichen.

8.3 Refaktorierung

Das Refaktorierungsprojekt wird auf Basis der in dieser Ausarbeitung entwickelten Richtlinien[3] angepasst. Diese Refaktorierung kann grob in drei wesentliche Bereiche

[1] Auf der beigefügten CD im Ordner **Quellcode_RefaktorierungsProjekt_NEU** zu finden
[2] Auf der beigefügten CD im Ordner **Quellcode_LagerscannerProjekt_ALT** zu finden
[3] Diese sind im Anhang unter A.7 auf Seite 75 zu finden

geteilt werden, welche nacheinander abgearbeitet werden und teilweise aufeinander aufbauen:

- Verbesserung der Wartbarkeit und Darstellung
- Verbesserung der Dokumentation
- Verbesserung der Fehlerbehandlung

Es werden dabei Beispiele für die einzelnen durchgeführten Änderungen präsentiert, wobei Ausschnitte der Quellcodes aus beiden Projekten gegenübergestellt und verglichen werden. Eine zusammenfassende Bewertung dieser Vergleiche wird in Kapitel 8.4.1 auf Seite 56 durchgeführt.

8.3.1 Verbesserung der Wartbarkeit und Darstellung

Die in der Analyse festgestellten Schwachstellen des Quellcodes werden im Folgenden gemäß der definierten Richtlinien überarbeitet und behoben. Dabei werden die Richtlinien zur Verbesserung der Wartbarkeit und Darstellung gemeinsam betrachtet und umgesetzt.

Entfernen von ungenutztem Quellcode

Eine erste einfache Maßnahme, um die Übersichtlichkeit des Quellcodes zu verbessern, ist das Entfernen von überflüssigen Kommentaren, die alten und in der aktuellen Version unbenutzten Quellcode beinhalten.

Variablen und Methoden, die von der Entwicklungsumgebung, sowie von SonarQube und auch Stylecop als obsolet angegeben wurden, werden entfernt.

An Stellen, die einer Variable das Ergebnis eines Methodenaufrufs zuweisen, die Variable dann aber nicht weiterverwenden, werden die Variablen entfernt und die Methodenaufrufe ohne eine Zuweisung durchgeführt. In den Quellcodes 8.1 auf Seite 48 sowie 8.2 auf Seite 49 sind die Auswirkungen davon beispielhaft zu sehen.

```
int result = tbls.Add();
string error =
    ↪ DIConnector.GetCompanyObject().GetLastErrorDescription();
System.Runtime.InteropServices.Marshal.ReleaseComObject(tbls);
tbls = null;
GC.Collect();
```

Quellcode 8.1: Zuweisungen auf ungenutzte Variablen

```
tbls.Add();
System.Runtime.InteropServices.Marshal.ReleaseComObject(UserTables);
GC.Collect();
```

Quellcode 8.2: Entfernen von ungenutzten Variablen

Aus allen Klassen werden ungenutzte Methoden entfernt. Wurden diese im Zuge einer Anpassung der Erweiterung angelegt, sind jedoch seitdem obsolet geworden und nicht aus dem Code entfernt worden, können sie nun aus dem Quellcode beseitigt werden. Eine Klasse beispielsweise beinhaltet eine Methode , deren einzige Funktion ist, eine andere Methode aus der gleichen Klasse aufzurufen. Der Inhalt dieser Methode wird in die aufrufende Methode verschoben und der Methodenrumpf wird anschließend entfernt.

Import-Statements vereinheitlichen

Aufrufe von importierten Objekten und Funktionen innerhalb der Klassen werden vereinfacht. Dazu wird in jeder Klasse festgestellt, welche Import-Statements für die Bereitstellung von externen Bibliotheken genutzt werden. Lange Import-Statements, wie beispielsweise in Quellcode A.2 auf Seite 74 zu sehen sind, werden durch den Einsatz von Aliasnamen vereinfacht.

```
using SwissAddonFramework.UI.Components;
using SwissAddonFramework.UI.EventHandling.ItemEvents;
```

Quellcode 8.3: Lange Import-Statements

In Quellcode 8.4 auf Seite 49 ist dies beispielhaft zu sehen. So wird sichergestellt, dass innerhalb der Quellcodes eindeutig festgelegt ist, aus welcher Bibliothek ein Objekt oder eine Funktion kommt.

```
using Components = SwissAddonFramework.UI.Components;
using Events = SwissAddonFramework.UI.EventHandling.ItemEvents;
```

Quellcode 8.4: Alias für lange Import-Statements

Zuordnung von Elementen verdeutlichen

Um die Zugehörigkeit von Elementen[4] zu Klassen zu verdeutlichen, wird jeder Aufruf eines Elementes einer Klasse darin mit *this.* ausgestattet. Dies betrifft alle Elemente, die innerhalb einer Klasse definiert und dort verwendet werden.

Variablen, die als Klassenvariablen definiert wurden, allerdings nur im Kontext einer Methode genutzt werden, werden aus der Klasse entnommen und innerhalb der betreffenden Methode lokal definiert und initialisiert. An dieser Stelle muss darauf geachtet

[4] Objekte, Variablen, Methoden

werden, dass die Variable nicht in anderen Komponenten oder Methoden verwendet wird. Die UI-Elemente aus SAP B1 beispielsweise müssen immer als globale Variablen angelegt werden.

Bilden langer Strings

In dem Refaktorierungsprojekt werden an manchen Stellen sehr lange Strings[5] gebildet, diese werden häufig allerdings nur einmalig genutzt und können durch die Aufteilung auf mehrere Zeilen soweit übersichtlich gemacht werden, dass der Einsatz von Stringbuildern an dieser Stelle übermäßig wäre.

In mehreren Fällen werden längere zusammengesetzte SQL-Abfragen innerhalb von Schleifen erzeugt, diese werden entsprechend der Richtlinien umgebaut, so dass sie durch Stringbuilder erzeugt werden. Der Einsatz von Stringbuildern für diese SQL-Abfragen sorgt dafür, dass die Methoden um einige Zeilen länger und unübersichtlicher wurden, weshalb die Entscheidung getroffen wurde, alle Einsätze von Stringbuildern in eigene Methoden einer statischen Klasse auszulagern.

Anordnung von Elementen einer Klasse

Um eine Vereinheitlichung der Klassenstrukturen zu schaffen, wurden alle Klassen gemäß der Grafik A.5 auf Seite 75 im Anhang angepasst, um alle Klassen in ihrem strukturellen Aufbau dem C#-Standard anzunähern.

Schlecht wartbare Programmierung verbessern

In der Analyse des Projektes fielen vor allem unintuitive Ausdrücke und Programmierungen als Schwachstellen der Wartbarkeit auf.

Beispielsweise wurden so die Vergleiche einer Methode unnötig komplex, hier wurden invertierte Operatoren genutzt, anstatt einfach gegenteilige Vergleichsoperatoren einzusetzen. In den Quellcodes 8.5 auf Seite 50 und 8.6 auf Seite 51 ist zu erkennen, wie sich die Lesbarkeit der Zeile vereinfachen lässt, wenn korrekte Operatoren für die einzelnen Vergleiche gewählt werden.

```
if (!(activeRight <= 0)
&& !(activeRight >= rightsGrid.Rows.Count)
&& !(activeUser.Equals("")))
```

Quellcode 8.5: Schwer verständliche Nutzung von Operatoren

[5] Für die Erzeugung von langen SQL-Abfragen

```
if (ActiveRight > 0
&& ActiveRight < RightsGrid.Rows.Count
&& !String.IsNullOrEmpty(ActiveUSer))
```

Quellcode 8.6: Verbesserte Anwendung von Operatoren

In einigen Vergleichen wurden unnötige Vergleiche von *Booleans* eingesetzt, wenn eine Variable auf ihren Wert überprüft werden sollte. Diese Vergleiche werden vereinfacht, sodass nur der *Boolean*-Ausdruck vorhanden ist. In den folgenden Beispielen der Quellcodes 8.7 auf Seite 51 und 8.8 auf Seite 51 ist eine solche Anpassung zu sehen.

```
if (oCompany.Connected == true)
{
        oCompany.Disconnect();
}
```

Quellcode 8.7: Unnötiger Vergleich mit Boolean

```
if (Company.Connected)
{
        Company.Disconnect();
}
```

Quellcode 8.8: Einfacherer Vergleich mit Boolean Wert

Weiterhin sind an einigen Stellen unnötige Klammern in Ausdrücken oder Zuweisungen zu finden, diese werden schlichtweg entfernt, solange sie keine Auswirkungen auf die Funktion des Codes haben. In den Quellcodes 8.9 auf Seite 51 und 8.10 auf Seite 51 ist eine solche Anpassung beispielhaft zu sehen.

```
this.userGrid =
    ↪ ((SAPbouiCOM.Grid)(this.GetItem("UserGrid").Specific));
```

Quellcode 8.9: Übermäßige Nutzung von Klammern

```
this.userGrid = (SAPbouiCOM.Grid)this.GetItem("UserGrid").Specific;
```

Quellcode 8.10: Entfernen von überflüssigen Klammern

Anpassung der Namensschemata

Der nächste Schritt befasst sich mit der Vereinheitlichung von Bezeichnern und der Anpassung an den durch Microsoft vorgeschlagenen Standard für C#. [18] Dabei werden die Klassen der Erweiterung nach Tabelle 8.1 auf Seite 52 umbenannt, um sowohl Pascalcase zu nutzen, als auch intuitive und sprechende Namen zu erhalten, um auf den ersten Blick erkennen zu können, worum es sich bei einer jeweiligen Klasse handelt.

Alter Klassenname	**Neuer Klassenname**	**Begründung**
frmRights	PermissionForm	Pascalcase und aussagekräftigerer Objekt-Name
frmLic	LicenseForm	Pascalcase und aussagekräftigerer Objekt-Name
Program	Program	Keine Änderung, Einstiegspunkt der Erweiterung ist von SAP vorgegeben
Menu	MainMenu	Genauere Beschreibung der Klasse für die Erstellung des SAP Menus
DIConnector	DatabaseConnector	Ungarische Notation entfernt
Database	DatabaseTables	Statische Klasse nach Hauptaufgabe benannt

Tabelle 8.1: Umbennung der Klassennamen

Ähnlich wie die Klassennamen, werden auch die Namen von Variablen, Methoden und Parametern an die Vorgaben der C#-Konventionen angepasst und gemäß der festgestellten Groß- und Kleinschreibung und sprechenden Namen, umbenannt.

Um weitere Vereinheitlichungen zu erzeugen, werden alle Bezeichner für Elemente des Quellcodes, auf englisch und gemäß ihrer Funktion, sprechend benannt. Gegebenenfalls werden diese dahingehend angepasst.

Typ	**Richtlinien zur Benennung**	**Beispiel**
Namespace	Beschreibung des Bereiches der Erweiterung	WarehouseAddon
Klasse	Nomen in Pascalcase	LicenseForm
Methode	Verben in Pascalcase	ReadLicense()
Eigenschaft oder Variable	Nomen oder Adjektiv in Pascalcase	Date, MaxUsers
Parameter	camelcase verwenden, abhängig von Methode	value, dateNumber
Exception	Beschreibende Phrase in Pascalcase mit Exception enden	IllegalDateEnteredException

Tabelle 8.2: Richtlinien für Bezeichner

In der Tabelle 8.2 auf Seite 52 ist zu erkennen, nach welchen Schemata die Bezeichner angepasst werden. Besonders die Nutzung der Ungarischen Notation und auch die

Anwendung von snake_case wird dabei vermieden und durch sprechende Bezeichner in PascalCase ausgetauscht. In den folgenden Quellcodes[6] sind in zwei Bezeichnern von Variablen sowohl die Anwendung von ungarischer Notation[7], sowie snake_case[8] zu sehen. Im Refaktorierungsprojekt werden diese Bezeichner durch Richtlinien-konforme Namen ersetzt.

```
private SAPbouiCOM.Button button_ok;
private SAPbouiCOM.Button button_ca;
```

Quellcode 8.11: Ungarische Notation

```
private Button OkButton;
private Button CancelButton;
```

Quellcode 8.12: Verbesserte Benennung

8.3.2 Verbesserung der Dokumentation

Nachdem im vorherigen Schritt ein Teil der Schwachstellen innerhalb des Quellcodes behoben wurde und Wartbarkeitsaspekte angepasst wurden, folgt eine ausführlichere und vollständige Dokumentation des Quellcodes auf Basis der bisher verbesserten Version. Dazu werden die in Kapitel 7.3.3 auf Seite 45 genannten Richtlinien zur Dokumentation genutzt.

Dokumentation von Klassen

Gemäß den Richtlinien werden die Klassen um Kopfkommentare in Form von XML-Dokumentationskommentaren [15] ergänzt. Dabei wird innerhalb der $<$ *summary* $>$-Tags festgehalten, was die Hauptaufgabe der Klasse ist und auf eventuelle Besonderheiten eingegangen. Die $<$ *remarks* $>$-Tags werden hingegen an dieser Stelle genutzt, um das Änderungsprotokoll festzuhalten. Änderungen am Quellcode der Klasse werden nach folgendem Muster dokumentiert: „Datum der Änderung - Name des zuständigen Entwicklers - Zusammenfassung der Änderungen“. Die Zusammenfassung der Änderungen wird anschließend bei der Nutzung der Versionskontrolle beim Einchecken der Änderungen in das Projektrepository als Nachricht eingetragen, um so Informationen zu den genauen Änderungen leicht finden zu lassen.

Ein vollständiger Dokumentationskopf einer Klasse sieht beispielsweise wie in Abbildung 8.1 auf Seite 54 aus.

[6] 8.11 auf Seite 53 und 8.12 auf Seite 53
[7] Nutzung vom Präfix “button “
[8] Klein geschrieben und durch Unterstriche verbundene Wörter

```
/// <summary>
/// Hilfsklasse für das bilden und liefern langer oder komplizierter Sql-Queries.
/// Statische öffentliche Methoden lassen sich aus jeder anderen Klasse aufrufen,
///     um die generierten Queries dort nutzen zu können.
/// </summary>
/// <remarks>
/// Änderungsprotokoll:
/// 04.11.2019 - Jan Rehmeyer - Erstellung der Klasse
/// 05.11.2019 - Jan Rehmeyer - Hinzufügen von Methoden
///     GetSortRightsQuery und GetSortDirectionQuery
/// </remarks>
```

Abbildung 8.1: XML Dokumentationskommentar für Klassen

Dokumentation von Methoden

Ähnlich wie Klassen, werden auch Methoden um XML-Dokumentationskommentare ergänzt. Diese nutzen die, in den Richtlinien vorgeschlagenen, Tags zur Beschreibung von Inhalt und Aufgabe sowie Parametern und Rückgabewerten.

Durch den Einsatz des SAPBusinessOneSDK ist es notwendig, dass Methoden für die Abarbeitung von UI-Events ein bestimmtes Set an Parametern übergeben bekommen. Häufig kommt es vor, dass die Methoden zur Abarbeitung von Events diese Parameter allerdings nicht nutzen. In den XML-Dokumentationskommentare der betroffenen Methoden konnte dies allerdings gekennzeichnet und beschrieben werden, sodass weniger Verwirrung auftritt. Die verwendete Kennzeichnung im Quellcode ist: „Unbenutzt, nötiger Parameter für SAP B1 Events“.

Dokumentation von weiterem Quellcode

Alle Line-Kommentare, die laut Richtlinien übererklärend oder überflüssig sind, werden entfernt.

Im nächsten Schritt werden Stellen im Quellcode herausgesucht, die Line-Kommentare für das Verständnis benötigen. Diese Stellen fallen dadurch auf, dass Inhalte oder Zusammenhänge nicht auf den ersten Blick klar werden und auch für Entwickler, die das Thema kennen, vielleicht schwer zu verstehen sind. Hierbei wird besonders darauf geachtet, nur das nötigste zu kommentieren und vor allem zu erklären, warum die Stelle beachtenswert ist.

Die kommentierte Zeile in Abbildung 8.2 auf Seite 54 beispielsweise kann an ihrer Position innerhalb der Methode verwirrend wirken, hier kann mit einem Line-Kommentar sichergestellt werden, dass deutlich wird, warum diese Zeile dort steht.

```
// Erneute Initialisierung des Grids, um Anzeige der Form neu zu laden (nötig für UI-Ansicht)
InitializeActiveUserRights();
```

Abbildung 8.2: Line-Kommentar zur Erklärung von Codezeilen

```
 private void CalculateOpenLicenses(DataTable table)
{
        if(!table.IsEmpty)
        {
                // Eigentlicher Code der Methode
        }
        else{
                ShowWarning("Empty Table: " + table.GetUID());
                Form.Close();
        }
}
```

Quellcode 8.13: Prüfung der Parameter innerhalb von Methoden

Im Refaktorierungsprojekt ist dies für die Dokumentation von SQL-Abfragen notwendig, um zu erklären, welchen Zweck eine Abfrage innerhalb des Quellcodes erfüllt. Daher werden alle Vorkommen von SQL-Abfragen innerhalb des Codes mit einem Line-Kommentar versehen, der eine grobe Beschreibung der SQL-Abfrage enthält.

8.3.3 Verbesserung der Fehlerbehandlung

Als dritte Maßnahme zur Umsetzung der Richtlinien wird zum einen die Fehlervermeidung innerhalb der Erweiterung betrachtet, zum anderen soll versucht werden, die Behandlung von Ausnahmen zu vereinheitlichen. Die Richtlinien dafür sind in Kapitel 7.3.4 auf Seite 45 definiert.

Fehlerbehandlung innerhalb des Quellcodes

Um die Fehlerbehandlung zu verbessern, werden vor allem zwei Umstände betrachtet. Zum einen wird in Methoden, die Parameter übergeben bekommen, eine Prüfung dieser eingeführt. Dabei soll jede Methode eine eigene Verantwortlichkeit dafür erhalten, die ihr übergebenen Parameter auf gültige Wertebereiche, *null*-Werte oder korrekte Datentypen zu überprüfen. Gegebenenfalls wird eine Fehlerbehandlung durchgeführt, oder aber Informationen über die Art des Fehlers geliefert. In Quellcode 8.13 auf Seite 55 ist eine dieser Prüfungen beispielhaft zu sehen.

Als zweite Maßnahme werden stärkere Prüfungen innerhalb der Methoden eingebaut, die dafür sorgen, ausgehende Parameter oder Berechnungen innerhalb einer Methode genauer zu überwachen und im Fehlerfall Auskunft über die Ursache des Fehlers geben zu können. In Quellcode 8.14 auf Seite 56 ist ein Beispiel dafür zu sehen. Zusätzlich zu der ursprünglich einzigen Prüfung der $MenuUID$, werden eine Prüfung auf den Typen des Events $pVal$, sowie eine Prüfung, ob die betroffene Form bereits in SAP B1 existiert, eingefügt.

```
if (pVal.BeforeAction && pVal.MenuUID.Equals("menuScApp.lic") &&
    ↪ !FormExists("frmLic"))
{
        LicenseForm ActiveFormLicense = new LicenseForm();
        ActiveFormLicense.Show();
}
```

Quellcode 8.14: Verwendund genauerer Prüfungen im Code

```
catch (ArgumentException e)
{
        SAPbouiCOM.Framework.Application.SBO_Application.StatusBar.
        SetText("Rechte konnten nicht aktualisiert werden, Grund: "
            ↪ + e.Message, BoMessageTime.bmt_Short,
            ↪ BoStatusBarMessageType.smt_Error);
}
```

Quellcode 8.15: Verbesserte Ausnahmebehandlung

Ausnahmebehandlung innerhalb des Quellcodes

Bestehende Ausnahmefälle werden korrigiert, sodass mit den dabei abgefangenen Exceptions umgegangen werden kann und diese korrekt behandelt werden. Die meisten Exceptions, die in der aktuellen Version der Erweiterung abgefangen werden, werden schlichtweg nicht behandelt und bilden damit leere Blöcke, die nur dazu dienen, das Kompilieren der Erweiterung zu ermöglichen.

In der Refaktorierung werden zunächst allen Exceptions die Bezeichner „e“ zugewiesen, um diese schnell und eindeutig erkennen zu können. Anschließend wurden die Exceptions mit jeweilig passenden Ausgaben innerhalb der SAP B1 UI ausgestattet und an, für den Programmablauf kritischen, Stellen dahingehend umgebaut. Der Quellcode 8.15 auf Seite 56 zeigt eine erste Verbesserung. In dieser wird in einem Ausnahmefall mindestens eine aussagekräftige Fehlermeldung in SAP B1 ausgegeben.

8.4 Auswertung der Umsetzung

8.4.1 Vergleich

Das Refaktorierungsprojekt wird, nach Umsetzung der oben genannten Änderungen, einer Untersuchung durch die in der Analyse eingesetzten Werkzeuge unterzogen und mit der vorherigen Version aus Kapitel 6.4 auf Seite 26 verglichen.

Es wird betrachtet, wie sich die Anpassungen auf den Quellcode ausgewirkt haben und ob daraus eine Verbesserung der allgemeinen Codequalität zu erkennen ist. Um zu gewährleisten, dass die Erweiterung durch die Anpassungen des Quellcodes weiterhin gemäß ihrer Anforderungen funktioniert und korrekte Ergebnisse liefert, wird das

Refaktorierungsprojekt in der gleichen Demo-Umgebung wie die ursprüngliche Version ausgeführt und manuell auf korrekten Programmablauf getestet. Dabei konnten keine Fehler oder Abweichungen im Programmablauf festgestellt werden.

Auf dem Refaktorierungsprojekt wird eine erneute SonarQube-Analyse durchgeführt, dies erlaubt einen direkten, oberflächlichen Überblick über auffällige Verletzungen der C#-Code-Konventionen. [14] Weiterhin wird eine erneute Überprüfung mithilfe von Stylecop auf dem Quellcode ausgeführt.

Erkennbar ist, dass das ursprüngliche Analyse-Ergebnis von SonarQube mit der Refaktorierung verbessert werden konnte. Die Anzahl der Code Smells konnte von 77 auf 12 reduziert werden, wobei erwähnenswert ist, dass die verbleibenden 12 Code Smells aufgrund der Nutzung des SAPBusinssOneSDK nicht vermeidbar sind. Die Grafiken 8.3 auf Seite 57 sowie 8.4 auf Seite 57 zeigen die Auswertung vor und nach der Refaktorierung.

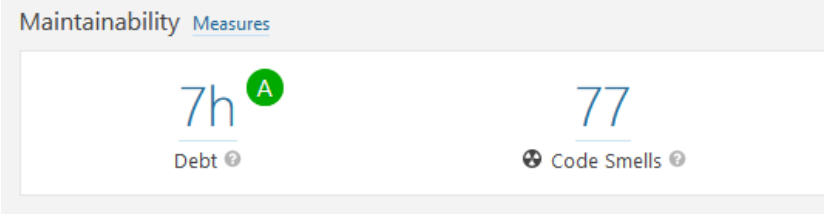

Abbildung 8.3: SonarQube-Wartbarkeit im Lagerscanner-Projekt

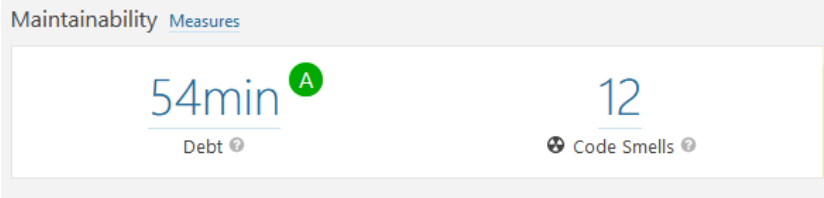

Abbildung 8.4: SonarQube-Wartbarkeit im Refaktorierungsprojekt

Zudem ist die Grafik 8.5 auf Seite 57 zu finden, diese beschreibt das schrittweise Entfernen von Code Smells und anderen Schwachstellen im Quellcode.

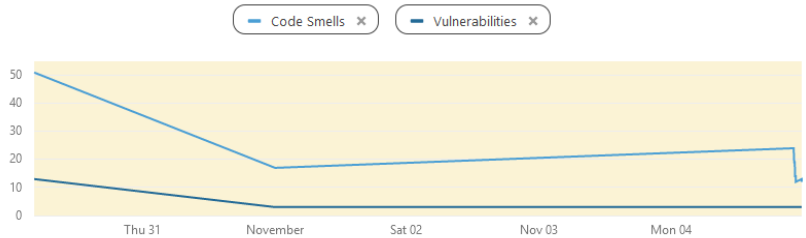

Abbildung 8.5: SonarQube-Aktivität Wartbarkeit

Die Vereinheitlichung der Namensschemata von Bezeichnern sorgt für eine einheitliche

Darstellung der Klassen, Methoden und Objekte innerhalb des Quellcodes und vereinfacht, unter Berücksichtigung des C#-Standards, die Lesbarkeit. In Kombination mit der, durch die Richtlinien vorgegebenen, Strukturierung der Klassen ist es nun möglich, auf den ersten Blick zu erkennen, was ein Element des Quellcodes ist, was seine Funktion ist, in welchem Kontext es zu verstehen ist und welche Zuordnung es innerhalb der Erweiterung besitzt.

Die Erweiterung konnte durch die Refaktorierung von 932 Zeilen Quellcode auf 892 Zeilen reduziert werden und ist einheitlicher und besser zusammenhängend. Betrachtet man die Anzahl der Kommentarzeilen, so fällt auf, dass diese von 170 (15.4%) auf 313 (26%) angestiegen sind. Eine Gegenüberstellung der Veränderungen der Codezeilen ist in Abbildung 8.6 auf Seite 58 zu sehen.

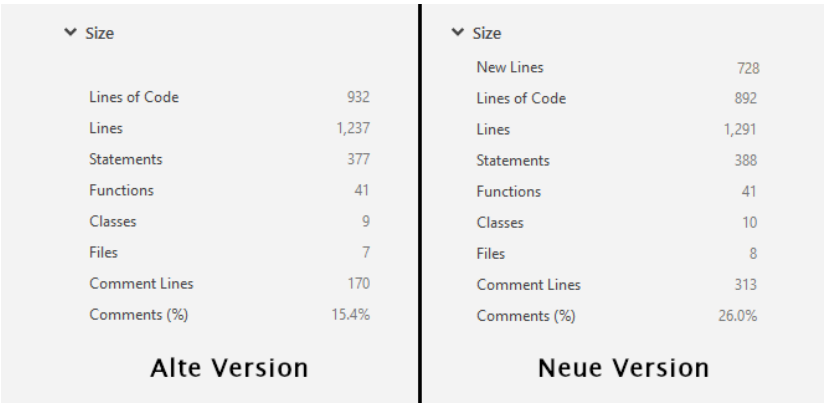

Abbildung 8.6: Lines of Code im Vergleich

Die eingesetzte Dokumentation beschreibt nun deutlicher, was die Aufgabe und der Kontext eines Objektes oder einer Methode sind. Zusätzlich werden innerhalb von Methoden nur an relevanten Stellen durch Line-Kommentare Unklarheiten beseitigt, was den Quellcode insgesamt leichter lesbar macht und seltener durch Kommentare auseinanderreißt. Die Einführung eines Bearbeitungsprotokolls im Kopfkommentar einer jeweiligen Klasse sorgt dafür, dass nicht der gesamte Inhalt einer Klasse durchsucht werden muss, um herauszufinden, an welchen Stellen im Quellcode Änderungen vorgenommen wurden.

Die Abbildung 8.7 auf Seite 58 zeigt die Entwicklung in der Anpassung der Dokumentation im Verlauf der Implementation.

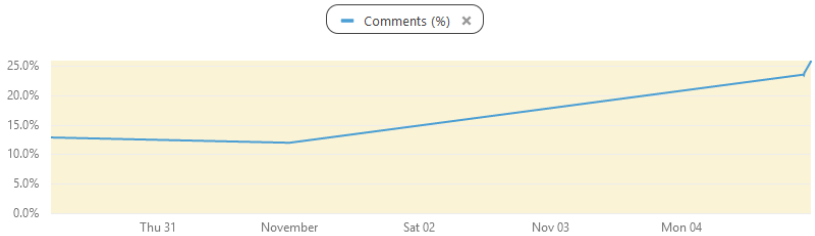

Abbildung 8.7: SonarQube-Aktivität Dokumentation

Aus dem Vergleich des Zustandes vor und nach der Implementation der Richtlinien ist zu erkennen, dass der Quellcode der Projekte besser wartbar[9] und einheitlicher geworden ist.

8.4.2 Abschließende Umfrage zur Umsetzung

Um einen Überblick über die Ergebnisse der Umsetzung der Richtlinien zu erhalten und um festzustellen, ob diese einen Nutzen für die Arbeit innerhalb der Abteilung liefern, wurde eine erneute Umfrage durchgeführt. Befragt wurden diesmal die drei verantwortlichen Entwickler, sowie zwei Berater, die an der Entwicklung von SAP B1 Erweiterungen beteiligt sind. Diese sollten sich vor der Umfrage mit dem Lagerscanner-Projekt und dem Refaktorierungsprojekt auseinandersetzen um Änderungen und mögliche Verbesserungen im Quellcode festzustellen.

Es wurde eine Frage zu den vier Teilbereichen der Richtlinien[10] gestellt, um herauszufinden, wie die Verbesserung dieser Aspekte durch die Anwendung der Richtlinien eingeschätzt wird.

Eine weitere Frage befasste sich mit der Verständlichkeit der erstellten Richtlinien.

Eine letzte Frage soll erörtern, inwieweit die Mitarbeiter eine langfristige Verbesserung der Softwarequalität von SAP B1 Erweiterungen innerhalb der Abteilung sehen.

Frage zur Qualitätsverbesserung der betrachteten Aspekte

Mit dieser Frage sollte versucht werden abzuwägen, in welchem Umfang die vier betrachteten Aspekte durch die Anwendung der Richtlinien verbessert werden konnten. Die befragten Mitarbeiter wurden gebeten, für jeden der Aspekte eine Bewertung von 0 bis 5 Punkten zu vergeben. Eine Punktzahl von 0 bedeutet, dass sich der Bereich durch die Refaktorierung nicht verbessert hat, eine Punktzahl von 5 bedeutet hingegen, dass das Projekt in dem genannten Bereich extrem verbessert wurde.

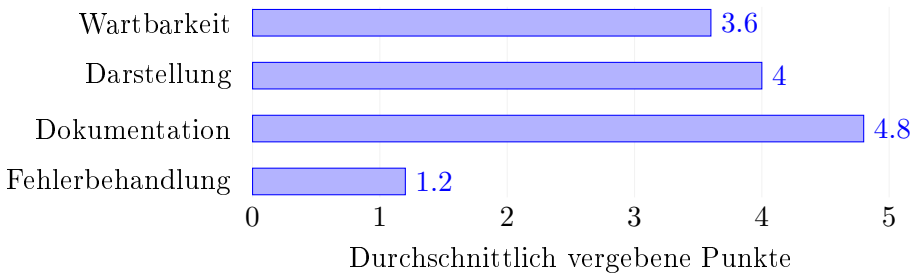

Abbildung 8.8: Antworten zur Qualitätsverbesserung der betrachteten Aspekte

[9] laut der SonarQube-Analyse
[10] Wartbarkeit, Darstellung, Dokumentation und Fehlerbehandlung

Abbildung 8.8 auf Seite 59 zeigt die durchschnittlich vergebenen Punkte. Es ist erkennbar, dass vor allem in der Darstellung und der Dokumentation starke Verbesserungen gesehen werden. Aber auch die Wartbarkeit wird nach Anwendung der Richtlinien als qualitativ hochwertiger eingeschätzt. Die Behandlung von Fehlern fällt am wenigsten stark auf, doch auch hier ist zu erkennen, dass nach Meinung der Befragten zumindest eine ansatzweise Verbesserung stattgefunden hat.

Frage zur Verständlichkeit der Richtlinien

Die befragten Mitarbeiter wurden gebeten, anzugeben, wie leicht verständlich und lesbar sie die Richtlinien in Form des User Manuals einschätzen. Ist die Lesbarkeit oder das Verständnis der Richtlinien eingeschränkt, so können vor der Einführung noch Anpassungen am User Manual durchgeführt werden, um dieses einfacher zu gestalten.

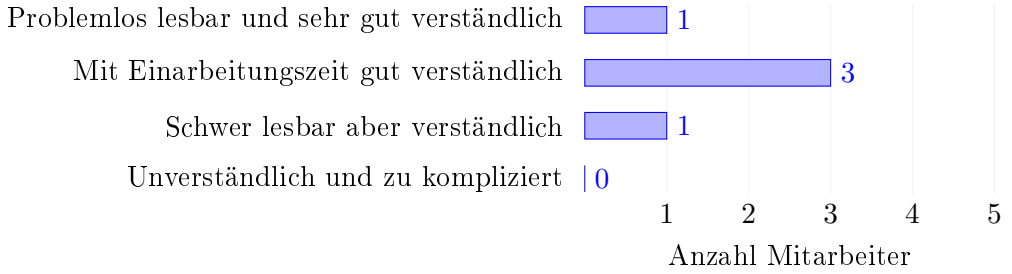

Abbildung 8.9: Antworten zur Verständlichkeit der Richtlinien

Die Abbildung 8.9 auf Seite 60 zeigt, dass die erstellten Richtlinien zwar verständlich, aber nicht für jeden befragten Mitarbeiter direkt umsetzbar sind. Mit einer Einarbeitungszeit sollte jedoch der Großteil der Befragten in der Lage sein, die Richtlinien in der Entwicklung umzusetzen.

Frage zur langfristigen Qualitätsverbesserung

Abschließend wurden die befragten Mitarbeiter gebeten, anzugeben, wie stark sie die Auswirkungen der Richtlinien auf eine langfristige Verbesserung der Softwarequalität innerhalb der Abteilung einschätzen.

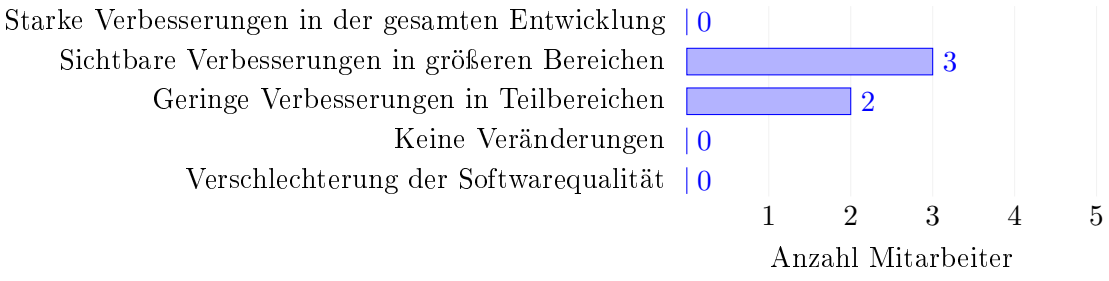

Abbildung 8.10: Antworten zur langfristigen Qualitätsverbesserung

Abbildung 8.10 auf Seite 60 zeigt, dass die befragten Mitarbeiter die Auswirkungen der Richtlinien in der Entwicklung als bemerkbar, aber nicht abschließend oder vollständig einschätzen. Eine Verbesserung der gesamten Qualitätssicherung innerhalb der Entwicklung der Abteilung zu erhalten ist allerdings auch im Rahmen dieser Ausarbeitung nicht das Ziel gewesen. Die Antworten liegen im erwarteten Bereich für diese erste Iteration.

8.4.3 Zusammenfassung

Sowohl die erneute Analyse und der darauf basierende Vergleich, als auch die Ergebnisse der Umfrage zeigen, dass die Umsetzung der Richtlinien durchaus eine Verbesserung der Qualität des Quellcodes in den betrachteten Projekten ergeben hat. Besonders der Umfang, aber auch die Qualität der Dokumentation des Quellcodes, sowie die Verbesserung der Lesbarkeit für die verantwortlichen Mitarbeiter lässt darauf schließen, dass die Richtlinien für die Arbeit in der SAP Business One Abteilung hilfreich und zielführend sind.

Es kann davon ausgegangen werden, dass ähnliche Projekte vergleichbare Ergebnisse liefern würden, sollten diese Richtlinien auf sie angewendet werden.

Wichtig zu beachten ist, dass diese beispielhafte Implementation nur Aussagekraft über ähnliche SAP B1 Erweiterungen hat. Für die Anwendung auf größere, oder sich stark unterscheidende Erweiterungen könnte eine weitere Prüfung der Richtlinien sinnvoll sein.

9 Auswertung

In diesem Kapitel werden die Ergebnisse dieser Ausarbeitung zusammengefasst. Außerdem wird beschrieben, wie die nächsten Iterationen der in Kapitel 3 auf Seite 9 beschriebenen Methodik aussehen könnten.

9.1 Fazit

Ein abschließender Überblick über die Ergebnisse zeigt, dass das Ziel der Ausarbeitung, Maßnahmen für die Verbesserung der Softwarequalität in Form von Richtlinien für die Entwicklung von SAP Business One Erweiterungen innerhalb der SAP Business One Abteilung zu schaffen, erreicht wurde.

Der Fokus dieser Ausarbeitung lag auf der Verbesserung von Softwarequalität nach einem zusammengesetztem Ansatz, basierend auf QIP und Modellen des ISO 9126 und Boehmes Softwarequalitätsmodells. Die Verbesserungen wurden hauptsächlich mit SonarQube und Stylecop 6.1 bewertet. Softwaretests wurden bewusst nicht betrachtet, da sie den Rahmen dieser Ausarbeitung gesprengt hätten.

Eine gesamte Analyse von sowohl Code- und Prozessqualität, als auch Architektur und Tests, wäre wünschenswert gewesen, wurde allerdings für den aktuellen Zustand der Abteilung als nicht zielführend erachtet. Ein kleinschrittiges Vorgehen und aufeinander aufbauende Maßnahmen zur Qualitätssicherung waren realistischer umsetzbar.

Die Entscheidung, einen Teilbereich der Softwarequalität, die Wartbarkeit des Quellcodes, zu betrachten, war richtig und für die aktuelle Situation innerhalb der Abteilung angemessen gewählt. Die Auswertung der Implementation in Kapitel 8.4 zeigt auf Basis der erneuten SonarQube-Analyse Verbesserungen innerhalb des Quellcodes und eröffnet potentielle Einstiegspunkte für zukünftige Iterationen und weitere Verbesserungsmaßnahmen, wie in Kapitel 9.2 beschrieben.

Insgesamt kann das Ergebnis der Ausarbeitung als nützlich und umsetzbar für die Abteilung bezeichnet werden und bietet eine gute Grundlage für die relativ unkonventionelle Softwareentwicklung von SAP B1 Erweiterungen innerhalb des Betriebes. Der starke Bezug auf die Arbeitsweise der Abteilung und die festgelegten Ziele aus Kapitel 5 auf Seite 21 schränkt die Nutzung der erstellten Richtlinien außerhalb des Betriebes natürlich stark ein. Dafür ist diese allerdings in der weiteren Entwicklung von SAP B1 Erweiterungen innerhalb der SAP Abteilung nutzbar. Es konnten zwar nicht alle der

genannten Qualitätsziele aus Kapitel 5 auf Seite 21 in ihrer Gesamtheit erfüllt werden, doch die allgemeine Zielsetzung, einen Anfang und eine Grundlage für Qualitätssicherung innerhalb der Abteilung zu beschreiben, konnte umgesetzt werden.

In Zukunft werden Veränderungen innerhalb der Entwicklung von SAP B1 Erweiterungen, wie auch in der Besetzung des Entwicklerteams innerhalb der Abteilung stattfinden. Potentielle personelle Erweiterungen sind nicht auszuschließen, führen zu Einarbeitungszeiten und neuen Arbeitsweisen. Durch den Einsatz der hier beschriebenen Richtlinien können diese neuen Mitarbeiter leichter in die Betrachtung und Bearbeitung der Quellcodes einsteigen.

9.2 Nächste Iterationen und Ausblick

Mit der beispielhaften Implementation wurde bewiesen, dass die betrachtete Software durch den Einsatz der erstellten Richtlinien qualitativ verbessert werden konnte. Um auf Kapitel 3.3: Zusammengesetzter Ansatz für den Betrieb auf Seite 10 zurückzukommen, sind die erstellten und getesteten Richtlinien, als Grundlage für weitere Iterationen des Ansatzes zu verstehen. Zusätzlich bilden sie eine Konsolidierung der in der Ausarbeitung erarbeiteten Ergebnisse.

Um Akzeptanz und Annahme der Richtlinien zu erreichen, werden diese nach einer abschließenden Vorstellung und darauf folgenden Schulungen innerhalb der Abteilung eingeführt. Danach können weitere Maßnahmen folgen, um eine fortlaufende Optimierung von Code und Prozessen im Betrieb zu schaffen. Dazu gehören unter anderem folgende Iterationen:

In einer Iteration sollte das Testen von Software in den Fokus gestellt werden. Um inhaltlich korrekte Software auszuliefern soll in dieser Iteration ein Konzept entwickelt werden, welches festlegt, wie und in welchem Umfang automatisierte Tests in der Entwicklung zum Einsatz kommen können.

Weitere Iterationen sollten sich auf in dieser Ausarbeitung nicht behandelte Merkmale der bisher betrachteten Modelle[1] konzentrieren und diese detaillierter betrachten.

Für spätere Iterationen ist es auch denkbar, weitere Modelle für die Erarbeitung von Zielen, Methodiken oder Merkmalen zu nutzen. Das in Kapitel 6.1 auf Seite 23 angesprochene Capability Maturity Model beispielsweise könnte langfristig betrachtet werden und als Grundlage einer späteren Iteration genutzt werden.

Den genauen Inhalt der nächsten Iterationen an dieser Stelle zu definieren ist noch nicht sinnvoll, da sich die Entwicklung innerhalb der SAP Business One Abteilung durch die Umsetzung der verschiedenen Iterationen verändern wird. Neue Entwickler bringen auch neues Wissen über Prozesse und Arbeitsweisen mit in den Betrieb, diese

[1] siehe dazu Kapitel 6.1 auf Seite 23

sollten aufgegriffen und in die Erweiterung oder Anpassung der Richtlinien und den darauf folgenden Iterationen des Verbesserungsvorgangs eingebracht werden.

Eine Verallgemeinerung der Ergebnisse für die Anwendung im gesamten Betrieb könnte ein langfristiges Ziel sein, um nicht nur innerhalb einer Abteilung, sondern im gesamten Betrieb ein einheitliches Entwicklungskonzept zu erhalten.

Literaturverzeichnis

[1] Al-Qutaish, Rafa E.: *Quality Models in Software Engineering Literature: An Analytical and Comparative Study.* – URL https://pdfs.semanticscholar.org/77b5/1002b53d002278997c71c72eaf2300a87ec7.pdf

[2] Al-Qutaish, Rafa E.: *Quality Models in Software Engineering Literature: An Analytical and Comparative Study.* – URL https://pdfs.semanticscholar.org/77b5/1002b53d002278997c71c72eaf2300a87ec7.pdf

[3] Coresystems: *Coresuite Homepage.* – URL https://coresystems.ch/coresuite-country-package/. – [Online; Letzter Zugriff am 13.11.2019]

[4] Dahlberg, CS: *Stylecop Repository.* – URL https://github.com/StyleCop/StyleCop. – [Online; Letzter Zugriff am 13.11.2019]

[5] Ghaffari, K.: *Magisches Dreieck: Kosten - Zeit - Qualität.* – URL https://www.gbcc.eu/3709/magisches-dreieck-kosten-zeit-qualitaet/. – [Online; Letzter Zugriff am 13.11.2019]

[6] GmbH, Capeletti & P.: *Homepage Capeletti & Perl GmbH.* – URL https://www.cpgmbh.de/. – [Online; Letzter Zugriff am 13.11.2019]

[7] KG, GOB Software & Systeme GmbH & C.: *Was ist ein ERP?.* – URL https://www.unitop-welt.de/landingpage/was-ist-erp/. – [Online; Letzter Zugriff am 13.11.2019]

[8] Liggesmeyer, Peter: *Ansätze zur Qualitätsverbesserung.* Kap. 1.3 Stand der Technik, S. 11. In: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*, Springer, 2009. – ISBN 978-3-8274-2056-5

[9] Liggesmeyer, Peter: *Kombinierte Ansätze.* Kap. 1.3 Stand der Technik, S. 13. In: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*, Springer, 2009. – ISBN 978-3-8274-2056-5

[10] Liggesmeyer, Peter: *Kontinuierliche Ansätze.* Kap. 1.3 Stand der Technik, S. 13–16. In: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*, Springer, 2009. – ISBN 978-3-8274-2056-5

[11] Liggesmeyer, Peter: *Modellbasierte Ansätze.* Kap. 1.3 Stand der Technik, S. 16–30. In: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*, Springer, 2009. – ISBN 978-3-8274-2056-5

[12] McCabe, Thomas J.: *A Complexity Measure*. – URL http://www.literateprogramming.com/mccabe.pdf

[13] Microsoft: *CA1800: Do not cast unnecessarily*. – URL https://docs.microsoft.com/en-us/visualstudio/code-quality/ca1800?view=vs-2019. – [Online; Letzter Zugriff am 19.11.2019]

[14] Microsoft: *Codekonventionen für C# (C#-Programmierhandbuch)*. – URL https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/inside-a-program/coding-conventions. – [Online; Letzter Zugriff am 18.11.2019]

[15] Microsoft: *Dokumentieren von Code mit XML-Kommentaren*. – URL https://docs.microsoft.com/de-de/dotnet/csharp/codedoc. – [Online; Letzter Zugriff am 19.11.2019]

[16] Microsoft: *Konventionen für die Groß-/Kleinschreibung*. – URL https://docs.microsoft.com/de-de/dotnet/standard/design-guidelines/capitalization-conventions. – [Online; Letzter Zugriff am 13.11.2019]

[17] Microsoft: *Namen von Klassen, Strukturen und Schnittstellen*. – URL https://docs.microsoft.com/de-de/dotnet/standard/design-guidelines/names-of-classes-structs-and-interfaces. – [Online; Letzter Zugriff am 18.11.2019]

[18] Microsoft: *Richtlinien für die Benennung*. – URL https://docs.microsoft.com/de-de/dotnet/standard/design-guidelines/naming-guidelines. – [Online; Letzter Zugriff am 19.11.2019]

[19] Microsoft: *SqlCommand.Prepare Methode*. – URL https://docs.microsoft.com/de-de/dotnet/api/system.data.sqlclient.sqlcommand.prepare?view=netframework-4.8. – [Online; Letzter Zugriff am 19.11.2019]

[20] Microsoft: *Ungarische Notation*. – URL https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa260976(v=vs.60)?redirectedfrom=MSDN. – [Online; Letzter Zugriff am 13.11.2019]

[21] Microsoft: *Vergleichsoperatoren (C#-Referenz)*. – URL https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/operators/comparison-operators. – [Online; Letzter Zugriff am 19.11.2019]

[22] Paulk, M.C.: *Capability maturity model, version 1.1*

[23] Rentrop, Christian: *Lesbarkeit von Quellcode erhöhen*. – URL https://www.dev-insider.de/lesbarkeit-von-quellcode-erhoehen-a-654539/. – [Online; Letzter Zugriff am 18.11.2019]

[24] Sami, Kokab: *Benefits of using SonarQube for Code Reviews.* – URL https://vizteck.com/blog/benefits-using-sonarqube/. – [Online; Letzter Zugriff am 13.11.2019]

[25] SAP: *SAP Business One Informationen.* – URL https://www.sap.com/germany/products/business-one.html. – [Online; Letzter Zugriff am 13.11.2019]

[26] SAP: *SAP Business One Test Composer.* – URL https://archive.sap.com/documents/docs/DOC-28659. – [Online; Letzter Zugriff am 13.11.2019]

[27] Skeet, Jon: *What are the default access modifiers in C#?.* – URL https://stackoverflow.com/questions/2917495/method-without-access-modifier. – [Online; Letzter Zugriff am 19.11.2019]

[28] Spolsky, Joel: *Making Wrong Code Look Wrong.* – URL https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/. – [Online; Letzter Zugriff am 19.11.2019]

[29] Warkentin, Nils: *SMART-Methode: Ziele richtig setzen und erreichen.* – URL https://karrierebibel.de/smart-methode/. – [Online; Letzter Zugriff am 15.11.2019]

[30] Wright, Jonathan: *Order of items in classes: Fields, Properties, Constructors, Methods - Antwort 1.* – URL https://stackoverflow.com/questions/150479/order-of-items-in-classes-fields-properties-constructors-methods. – [Online; Letzter Zugriff am 20.11.2019]

A Anhang

A.1 Magisches Dreieck der Software-Entwicklung

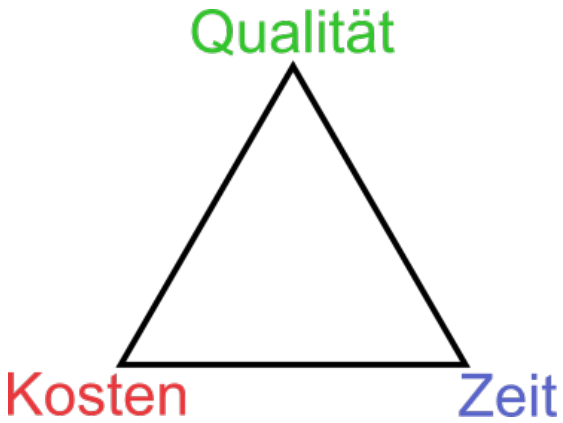

Abbildung A.1: Magisches Dreieck der Software-Entwicklung

A.2 Ansätze

Schritt	Aktion
1	Charakterisieren und Verstehen der Organisation und Umgebung
2	Definition quantifizierbarer Verbesserungsziele
3	Auswahl geeigneter Prozesse, Methoden, Techniken und Werkzeuge zur Umsetzung der Verbesserungsziele
4	Durchführung des definierten Projektes
5	Analyse der Resultate, Aufzeigen der Schwächen und Verbesserungsempfehlungen
6	Konsolidierung der gemachten Erfahrungen in Form von Modellen

Tabelle A.1: Quality Improvement Paradigm

A.3 Erläuterung von Qualitätsmerkmalen

Die Tabelle A.2 beschreibt die Merkmale des ISO 9126, während die Tabelle A.3 die Merkmale aus Boehmes Qualitätsmodell beschreibt.

Merkmal	**Erläuterung**
Änderbarkeit	Beschreiben die Wartbarkeit und wie leicht Änderungen an einem Programm vorgenommen werden können
Effizienz	Beschreibt das Verhältnis zwischen der Leistung des Programms und den eingesetzten Mitteln, diese zu erreichen
Übertragbarkeit	Gibt an, wie einfach es ist, das Programm in eine andere Umgebung zu übertragen
Zuverlässigkeit	Beschreibt, ob ein Programm eine Leistung über einen gewissen Zeitraum erhalten kann
Funktionalität	Gibt an, inwieweit ein Programm die geforderten Anforderungen erfüllt
Benutzbarkeit	Beschreibt den Aufwand, den ein Nutzer für die Benutzung eines Programms aufwenden muss

Tabelle A.2: Qualitätsmerkmale aus ISO 9126

Merkmal	**Erläuterung**
Flexibility	Möglichkeiten eines Programms auf Änderungen zu reagieren
Efficiency	Optimale Nutzung von Ressourcen durch das Programm
Portability	Das Programm kann auf verschiedenen Systemen ausgeführt werden
Reliability	Verlässlicher Programmablauf und akkurate Ergebnisse
Usability	Das Programm ist leicht zu benutzen
Understandability	Das Programm ist leicht zu verstehen
Testability	Möglichkeiten, das Programm zu testen sind vorhanden und die Ergebnisse sind aussagekräftig

Tabelle A.3: Qualitätsmerkmale aus Boehmes Qualitätsmodell

A.4 SonarQube Analyse-Ergebnisse

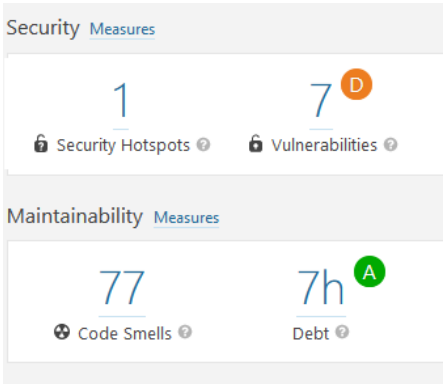

Abbildung A.2: Sonarqube Analyse-Ergebnisse für die Lagerscanner-Rechteverwaltung

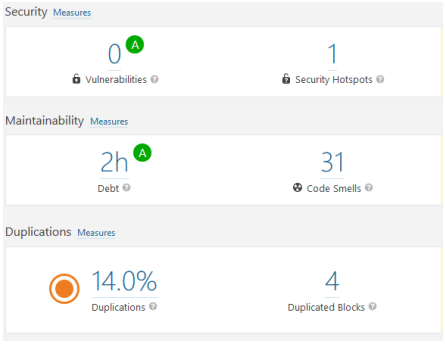

Abbildung A.3: Sonarqube Analyse-Ergebnisse für das Verknüpfungsplan-Projekt

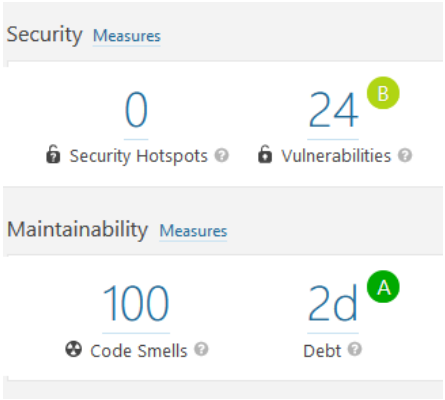

Abbildung A.4: Sonarqube Analyse-Ergebnisse für das Rechnungsgenerator-Projekt

A.5 Codebeispiele

```
using SAPbobsCOM;
using SAPbouiCOM;
```

Quellcode A.1: Typische Import-Statements

```
SAPbouiCOM.Framework.Application.SBO_Application.StatusBar.SetText("Text");
```

Quellcode A.2: Komplizierter Aufruf von importierten Objekten

```
 public void CreateMenu(MenuItem menuItemConfiguration)
{
//MenuItem sapModules = MenuItem.GetFromUID("43520");
//MenuItem menuICOF = MenuItem.CreateNew("Kistenmacher");
//MenuItem menuAssistent = MenuItem.CreateNew("Verknuepfungsplan");

//menuICOF.Type = MenuItem.MenuType.Popup;
//menuICOF.Value = "Kistenmacher";
////menuYuneec.Image = SwissAddonFramework.Global.BaseDirectory +
    ↪ @"\COR_Rz20Abrechnung\16x16transparent.bmp";
//sapModules.SubMenus.AddMenuItem(menuICOF);

//menuAssistent.Value = "Verknuepfungsplan";
//menuICOF.SubMenus.AddMenuItem(menuAssistent);

//menuAssistent.AddHandler_Click(delegate
    ↪ (SwissAddonFramework.UI.EventHandling.MenuEvents.MenuClick
    ↪ eventVal)
//{
//    Verknuepfungsplan v = new Verknuepfungsplan(FormUID);
//});

//sapModules.Load();
}
```

Quellcode A.3: Beispiel für auskommentierten alten Quellcode

A.6 Klassenstruktur

Abbildung A.5: Vorgeschlagene Struktur von Klassen

A.7 Benutzerhandbuch der Richtlinien

An dieser Stelle ist das in Kapitel 7 auf Seite 43 beschriebene Benutzerhandbuch zu finden.

Richtlinien für die Entwicklung von SAP Business One Erweiterungen

Jan Rehmeyer

Eine Sammlung von Richtlinien für die Verbesserung der Wart- und Lesbarkeit von Quellcode in SAP Business One Erweiterungen, zugeschnitten auf die Softwareentwicklung innerhalb der SAP Abteilung der Capeletti & Perl GmbH.

Inhaltsverzeichnis

Einleitung

Um innerhalb der SAP Abteilung eine Grundlage für die Sicherstellung von Softwarequalität in der Entwicklung von SAP Business One Erweiterungen zu gewährleisten, wurde im Rahmen einer Bachelorarbeit an der HAW Hamburg und in der SAP Abteilung der Capeletti & Perl GmbH dieses Benutzerhandbuch für die Entwickler der SAP Abteilung angelegt.
Es soll damit eine Basis geschaffen werden, die Begriffe Softwarequalität und Qualitäts-Sicherung in die Abteilung einzuführen und diese langfristig umzusetzen.

Dieses Benutzerhandbuch ist nicht vollständig und deckt im aktuellen Zustand nur einen Teilbereich der Softwarequalität ab.
Dies ist durchaus gewollt, das Ziel dieser Richtlinien und der dazugehörigen Ausarbeitung ist es, möglichst einfache Aspekte der Qualitätssicherung in der Abteilung umzusetzen und darauf aufbauend neue Konzepte zu entwickeln, die weiterführende Softwarequalität begünstigen. Die Richtlinien werden durch praktische Erfahrungen, Analyse-Ergebnisse und den Einsatz neuer Modelle erweitert und angepasst, um so langfristig eine Sammlung an Anforderungen, Prozessen und Regeln zu bilden, die für die Entwicklung von Softwareprojekten gültig sein soll.

Im Folgenden wird erklärt, wie dieses Dokument zu verstehen und anzuwenden ist. Weiterführend werden die einzelnen Bereiche kurz erläutert und anschließend die erarbeiteten Richtlinien erklärt.

Benutzung der Richtlinien

Die in den folgenden Kapiteln genannten Richtlinien sind als Vorgaben und Vorschläge für die zukünftige Entwicklung von SAP Business One Erweiterungen zu betrachten.
Dabei sollen diese dazu dienen, in der Entwurfs- und besonders in der Implementationsphase einer solchen Erweiterung, eine Orientierung zu geben, wenn es um die Qualitätskontrolle des Quellcodes geht.

Die Richtlinien befassen sich mit vier Bereichen, in denen relevante Merkmale und typische Schwachstellen aus den bisherigen Projekten vermerkt und ausgebessert dargestellt werden. Es werden hauptsächlich Aspekte der Wartbarkeit und der Darstellung von Quellcode betrachtet und einzeln aufgeführt, die einzelnen Richtlinien sind kurz und leicht umzusetzen, können aber in der Summe zu einer Verbesserung der Softwarequalität in den Projekten sorgen.

Wichtig zu beachten ist, dass sich diese Richtlinien in ihrer aktuellen Form ausschließlich auf SAP Business One Erweiterungen aus der Entwicklung der Abteilung beziehen. Codebeispiele sind in C# dargestellt und viele der Kennworte oder spezifischen Verbesserungsvorschläge beziehen sich auf Eigenheiten dieser Programmiersprache.
Die zugrundeliegenden Schwachstellen und Lösungen sind allerdings unabhängig von der Programmiersprache und können allgemein genutzt werden.

Insgesamt soll dieses Dokument dazu genutzt werden, bei Unklarheiten oder Schwierigkeiten in der Entwicklung von SAP Business One Erweiterungen, als Nachschlagewerk zu dienen, um einen Abteilungsübergreifenden Standard für die Erstellung von Quellcode zu erzeugen. Mit Annahme dieses Dokuments würde eine verpflichtende Wahrnehmung und Umsetzung der hier genannten Aspekte in Kraft treten.
Ausnahmen sind natürlich nicht auszuschließen, doch um sicherzustellen, dass schlussendlich eine stärkere Vereinheitlichung und Vereinfachung des Entwicklungsaufwandes entstehen kann, müssen sich alle verantwortlichen Entwickler daran halten.

Richtlinien zur Wartbarkeit

Unter den Richtlinien zur Wartbarkeit von Quellcode werden all jene Regeln und Vorgaben zusammengefasst, die sich mit der Verständlichkeit und Vereinheitlichung von Quellcode befassen. Viele der hier genannten Richtlinien sind darstellerischer Natur oder behandeln grobe Verletzungen der Lesbarkeit des Codes und sind daher auch vergleichsweise einfach zu erkennen und zu beheben. Bei komplexeren Richtlinien werden unterstützend Beispiele eingesetzt, die aufzeigen sollen wie Quellcode geschrieben werden kann.

3.1 Ungenutzte Methoden und Variablen

Ungenutzte Variablen und Methoden erfüllen keine Funktion innerhalb des Quellcodes und sorgen dafür, dass unnötiger Aufwand benötigt wird, um herauszufinden, warum sie existieren und womit sie zusammenhängen. Um dies zu vermeiden sind von der Entwicklungsumgebung als unbenutzt vermerkte Elemente des Quellcodes zu entfernen oder soweit in andere Bereiche zu integrieren, dass jedes Element des Quellcodes eine Funktion und Sinn besitzt.

Methoden oder Codeblöcke, die keinen Inhalt besitzen sollten nach Möglichkeit entfernt werden. An jenen Stellen, die zwar einen Methodenrumpf benötigen, aber keinen Inhalt besitzen müssen, um die Ausführung des Codes zu ermöglichen, sollten diese gekennzeichnet werden.
Folgende Kennzeichnung ist hierfür zu benutzen: „Ablaufrelevante Methode“.

3.2 Duplikate

Codeverdopplungen sind zu vermeiden. Diese werden standardmäßig durch die eingesetzte Entwicklungsumgebung markiert und sollen dementsprechend zusammengefasst oder in eigene Funktionen ausgelagert werden, die dann von den betroffenen Stellen aufgerufen werden kann. Ist dies nicht möglich so sollte in einem Line-Kommentar erklärt werden, warum dies der Fall ist.

3.3 Null-Werte

Um eindeutige Vergleiche mit null-Werten zu erhalten, werden folgende Zuweisungen und Vergleiche eingesetzt:

- Strings: Statt "" sollte *String.IsNullOrEmpty* sowie *String.Empty* genutzt werden.
- Objekte: Statt diese mit einem beliebigen Wert, oder gar nicht zu initialisieren, wird stattdessen *null* verwendet.
- Integer: Primitive Datentypen, wie beispielsweise *int*, können mit numerischen Null-Werten initialisiert und verglichen werden.

3.4 Vereinfachung von Ausdrücken

Überflüssige Klammern sind zu vermeiden. Besonders in Vergleichen kommt es immer wieder vor, dass Ausdrücke mit mehr Klammern versehen werden, als notwendig sind. In Quellcode 3.1 ist ein Beispiel dafür zu sehen:

```
if ((int)(Zahl + 1) != (Zahl + 2))
// Kann ersetzt werden durch:
if ((int)Zahl + 1 != Zahl + 2)
```

Quellcode 3.1: Entfernen von unnötigen Klammern

Besonders bei Vergleichen und Zuweisungen sollte darauf geachtet werden die korrekten Zeichen zu verwenden. Es soll immer der einfachste und natürlichste Weg gewählt werden, um diese Ausdrücke zu erstellen. In Quellcode 3.2 ist eine solche Anpassung zu sehen:

```
!(Zahl <= AndereZahl)
// Kann ersetzt werden durch:
Zahl > AndereZahl
```

Quellcode 3.2: Einsatz einfacherer Ausdrücke

3.5 Einsatz von Stringbuildern für SQL-Queries

Bei dem Erzeugen von großen Strings, wie beispielsweise langen SQL-Abfragen, sollte zur Verbesserung der Performance und Übersichtlichkeit ein Stringbuilder genutzt werden, anstatt die Teilstrings zu konkatenieren. Besonders innerhalb von Schleifen ist dies relevant, da potentiell viele Aufrufe die Performance der Erweiterung merklich beeinflussen können.

Um einen Stringbuilder verwenden zu können, muss die Bibliothek System.Text importiert werden.

Im folgenden Quellcode 3.3 wird ein einfaches Beispiel für den typischen Einsatz von Stringbuildern in SAP B1 Erweiterungen aufgeführt. Weitere Informationen können in der offiziellen .Net-Dokumentation der Bibliothek unter `https://docs.microsoft.com/de-de/dotnet/api/system.text.stringbuilder?view=netframework-4.8` nachgelesen werden.

```
int Zahl = 1
StringBuilder Builder = new StringBuilder();
Builder.Append("Beispiel Text: ");
Builder.Append(Zahl);
// Ausgabe durch Builder.ToString() ergibt dann:
   ↪ "Beispiel Text: 1"
```

Quellcode 3.3: Verwendung eines Stringbuilders

Um die Übersichtlichkeit weiterhin zu erhöhen, sollten lange SQL-Abfragen nach Möglichkeit in eigene Klassen ausgelagert werden, in denen sie mithilfe von Stringbuildern erzeugt und in den jeweiligen Klassen aufgerufen werden können.

3.6 Verschachtelungen

Verschachtelungen sollten nach Möglichkeit durch einfacher lesbare Strukturen ersetzt werden. So können beispielsweise mehrere, verschachtelte if-Abfragen durch Switch-Anweisungen ersetzt, oder ineinander integriert werden. Diese ineinander verschachtelten Blöcke können beispielsweise durch folgende Vereinfachung ersetzt werden:

```
if(Zahl == 1){
        if(AndereZahl == 0){
                ...
        }
}
// Kann ersetzt werden durch:
if(Zahl == 1 && AndereZahl == 0){
...
}
```

Quellcode 3.4: Verschachtelungen

Try-Blöcke in Try-Blöcken können auf die gleiche Art häufig ineinander integriert werden und sollten nach Möglichkeit angepasst werden. Ausnahmen hierbei bilden Fälle, in denen unterschiedliche Exceptions für mehrere Fälle erwartet und unabhängig voneinander abgearbeitet werden müssen.

3.7 Import-Statements

Um die Lesbarkeit des Codes zu erhöhen sollten Imports für häufig verwendete Elemente einen möglichst genauen Pfad im *using* Statement am Anfang einer jeden Klasse nutzen. Damit können in den Aufrufen der importierten Bibliotheken unnötig lange Pfade vermieden werden. Es ist allerdings darauf zu achten, dass trotzdem alle importierten Objekte korrekt aufgerufen werden können.
Import-Statements sind nur dann einzusetzen, wenn die importierten Bibliotheken oder Teile daraus in der Klasse benötigt werden. Durch die Entwicklungsumgebung als unbenutzt angezeigte Imports sind aus dem Code zu entfernen.
Sollten mehrere Import-Statements aus der gleichen Überliegenden Bibliothek nötig sein, so kann das jeweils längere Statement durch ein Alias ersetzt werden.

```
using Framework = SAPbouiCOM.Framework;
```

Quellcode 3.5: Nutzung von Alias für Import-Statements

3.8 Eindeutige Zuordnung von Objekten

Objekte einer Klasse sollen beim Aufruf innerhalb dieser Klasse immer mit dem Kennwort *this.* versehen werden, um eine absolut eindeutige Zuordnung der Klassenobjekte zu gewährleisten. Dadurch werden diese Aufrufe zwar länger, sorgen aber für eine einfachere Unterscheidung von Klassenvariablen und anderen Objekten.

Klassenvariablen, die innerhalb einer Klasse definiert werden, aber dort nur von einer einzigen Methode genutzt werden, werden in die betreffende Methode verschoben und dort als lokale Variable definiert. Globale Klassenvariablen sollten vermieden werden, um den Umfang einer Klasse möglichst gering und somit übersichtlich zu halten.

Richtlinien zur Darstellung

Um das vorhergehende Kapitel zu unterstützen und weitere Verbesserungen an der Darstellung des Quellcodes vornehmen zu können, befasst sich dieses Kapitel mit der Darstellung des Codes. Hiermit sind Richtlinien gemeint, die keine direkten inhaltlichen Auswirkungen auf den Quellcode haben, sondern lediglich dazu dienen, diesen kompakter oder einfacher zu gestalten. Zusätzlich werden an dieser Stelle vereinheitlichende Vorgaben für die Benennung von Bezeichnern für alle Elemente des Quellcodes vorgegeben.

4.1 Anordnung der Elemente einer Klasse

Die Reihenfolge der Elemente einer Klasse sollte stets einer vorgegebenen Reihenfolge entsprechen, um die Übersichtlichkeit zu erhöhen. So kann jeder Entwickler, der sich ein Projekt anschaut, direkt erkennen, wo welches Element einer Klasse zu finden ist. Die Struktur einer Klasse wird folgendermaßen vereinheitlicht:

Abbildung 4.1: Vorgeschlagene Struktur von Klassen

4.2 Benennung von Bezeichnern

Im Zuge der Vereinheitlichung von Quellcode sollen alle Bezeichner strikt nach dem C#-Namensschema benannt werden. Dabei sind folgende Vorgaben zu beachten:
Die Groß- und Kleinschreibung aller Bezeichner (ausgenommen Parameter) ist nach dem Schema des PascalCase vorzunehmen. Dabei wird der erste Buchstabe eines Wortes groß, das restliche Wort klein geschrieben. Bei zusammengesetzten Worten, wie zum Beispiel PascalCase wird auch jeweils der erste Buchstabe des angehängten Wortes groß geschrieben. Parameter bilden dabei eine Ausnahme, diese werden nach camelCase benannt. Diese Art der Groß- und Kleinschreibung unterscheidet sich von PascalCase dahingehend, dass der erste Buchstabe eines Bezeichners immer klein geschrieben wird. Bei zusammengesetzten Wörtern wird weiterhin der erste Buchstabe der angehängten Wörter groß geschrieben.
Allgemein gilt weiterhin:

- Die Sprache, in der programmiert wird, ist englisch, daher erhalten Bezeichner englische Namen.
- Es sollen keine Präfixe für Bezeichner genutzt werden.
- Es sollen keine Unterstriche für Bezeichner genutzt werden, stattdessen können zusammengesetzte Namen gewählt werden.

Da die Bezeichner sprechend und aussagekräftig sein sollen, gibt es Vorgaben, wie die folgenden Objekte zu benennen sind:

Typ	**Richtlinien zur Benennung**	**Beispiel**
Namespace	Beschreibung des Bereiches der Erweiterung	WarehouseAddon
Klasse	Nomen in Pascalcase	LicenseForm
Methode	Verben in Pascalcase	ReadLicense()
Eigenschaft oder Variable	Nomen oder Adjektiv in Pascalcase	Date, MaxUsers
Parameter	Pascalcase verwenden, abhängig von Methode	value, dateNumber
Exception	Beschreibende Phrase in Pascalcase mit Exception enden	IllegalDateEnteredException

Tabelle 4.1: Richtlinien für Bezeichner

4.3 Alter Code in Kommentaren

Alter und unbenutzter Code, der auskommentiert oder anderweitig unbrauchbar gemacht wurde, ist aus dem Quellcode zu entfernen. Um weiterhin Zugriff auf alten Code zu haben, wird die Versionskontrolle genutzt, um alte Versionen der Erweiterungen zu betrachten.

4.4 Spacing und Leerzeilen

Zwischen Operatorzeichen und Variablen oder Werten sollte in Vergleichen und Zuweisungen genau ein Leerzeichen enthalten sein, um die Übersichtlichkeit zu wahren. Siehe Quellcode 4.1 für ein Beispiel:

```
int Number = 1;
if(Number <! 0){...}
```

Quellcode 4.1: Spacing im Quellcode

Zudem sollten unnötige Leerzeilen oder komplett fehlende Leerzeilen zwischen Elementen einer Klasse vermieden werden. Zwischen den in Grafik 4.1 beschriebenen Klassenelementen sollten jeweils genau eine Leerzeile stehen, genauso wie zwischen jeder Methode innerhalb eines jeweiligen Blockes genau eine Leerzeile für die visuelle Trennung sorgen soll.

Richtlinien zur Dokumentation

Um einheitliche und aussagekräftige Dokumentation zu ermöglichen, werden in diesem Teil der Richtlinien Vorgaben und Vorschläge zu der Dokumentation von Klassen, Methoden und Variablen gemacht. Weiterhin wird erklärt, an welchen Stellen Inhalte wie Bearbeitungsprotokolle, Dokumentationsköpfe oder Erklärungen zu Codezeilen einzusetzen sind.
Allgemein sollen für die Dokumentation von SAP B1 Erweiterungen XML-Dokumentations-Kommentare genutzt werden.
Die XML-Dokumentationskommentare sind nach folgendem Schema, zu sehen in Quellcode 5.1, aufgebaut und können durch den Einsatz verschiedener Tags erweitert werden, um beispielsweise Informationen zu Parametern, Rückgabetypen, Ausnahmen oder Verweisen zu beinhalten.

```
/// <summary>
/// Beschreibung der Methode
/// </summary>
/// <remarks>
/// Weitere Informationen
/// </remarks>
/// <param name="text"> Beschreibung der Parameter
    ↪ </param>
/// <returns> Beschreibung der Rueckgabewerte </returns>
public int Function(string text){
        ...
}
```

Quellcode 5.1: XML-Dokumentationskommentar

Hauptsächlich werden die folgenden Tags zum Einsatz kommen (Tabelle 5.1), diese decken die vorkommenden und zu dokumentierenden Fälle ab:

Tag	**Beschreibung**
<summary>	Fasst die Aufgabe und den Inhalt eines Objektes oder einer Methode zusammen.
<param>	Beschreibt den Datentyp, Namen und die Aufgabe eines Parameters einer Methode. Pro Parameter muss ein weiterer dieser Tags angelegt werden.
<returns>	Beschreibt den Datentyp und Aufgabe des Rückgabewertes einer Methode.
<remarks>	Ergänzende Informationen einer Klasse im Klassenkopf.

Tabelle 5.1: Tags für XML-Dokumentationskommentare

5.1 Dokumentation von Klassen

Klassen sollen hauptsächlich durch einen sogenannten Klassenkopf in ihrer Funktion und ihrem Kontext beschrieben werden. Weitere Dokumentation der einzelnen Elemente einer Klasse wird direkt an diesen durchgeführt. Der Kopfkommentar einer Klasse beinhaltet folgende Informationen:

- Hauptaufgabe der Klasse
- Kontext im Gesamtprojekt
- Besonderheiten
- Bearbeitungsprotokoll

Diese Informationen werden mithilfe von XML-Dokumentationskommentaren, unter dem Einsatz von <summary>- sowie <remarks>-Tags, festgehalten.

5.2 Bearbeitungsprotokolle

Bearbeitungsprotokolle sind nicht direkt im Quellcode an den bearbeiteten Zeilen zu vermerken, sondern an einem zentralen Ort innerhalb der betroffenen Klasse, dem Dokumentationskopf. Dabei wird der Zeitpunkt der Änderung, der verantwortliche Mitarbeiter, sowie die durchgeführten Änderungen, beschrieben. Um die durchgeführten Änderungen im Quellcode wiederzufinden, wird die eingesetzte Versionskontrolle genutzt. Dieser soll bei Einchecken der Änderungen die gleiche Nachricht aus dem Bearbeitungsprotokoll beigefügt werden, um so eine Zuordnung herstellen zu können. Innerhalb der Versionskontrolle können nun Änderungen nachvollzogen werden, ohne dass unnötige Kommentare den Code füllen und von Hand eingetragen oder entfernt werden müssen.

5.3 Dokumentation von Methoden

Methoden sollten, ähnlich wie Klassen einen Kopfkommentar erhalten, in dem festgehalten wird, was die jeweilige Methode für eine Aufgabe hat und was durch diese Methode bearbeitet wird. Dies sorgt für eine Einordnung im Kontext der Anwendung und besitzt Aussagekraft über die Rolle und Relevanz der Methoden.
Entscheidend ist hierbei, zu beschreiben, was die Aufgabe einer Methode ist. Wie sie funktioniert soll durch sprechenden und klar strukturierten Code gegeben werden.
Zusätzlich sollen Eingabeparameter, sowie ihre gültigen Bereiche und Werte beschrieben werden. Schlussendlich werden noch die Rückgabewerte und deren gültige Werte, beziehungsweise Bereiche beschrieben, sowie, wozu diese Rückgabewerte dienen sollen. Um dies klar und übersichtlich darzustellen, werden auch hier XML-Dokumentationskommentare eingesetzt, hierbei kommen in jedem Fall der Tag <summary>, sowie, abhängig von den Parametern und Rückgabewerten der Methoden, die Tags <param> und <returns> zum Einsatz. Der Tabelle 5.1 sind die Funktionen dieser Tags zu entnehmen.

5.4 Dokumentation von Variablen

Variablen sollen gemäß den Richtlinien zur Benennung von Objekten sprechende und aussagekräftige Namen besitzen, die Informationen zu Typ und Funktion liefern sollten und müssen daher nicht gesondert mithilfe von Kommentaren dokumentiert werden. Ausnahmen bilden dabei selten benutzte, oder für den Ablauf der Erweiterung elementare Objekte, wie beispielsweise SAP B1 Objekte mit wenig aussagekräftigen Datentypen oder Namen.

5.5 Weitere Dokumentation von Quellcode

Besonders bei komplexeren Algorithmen oder Berechnungen kann es vorkommen, dass auch sprechender Code nicht ausreicht, um zu beschreiben, was genau eine Funktion oder Methode macht, oder wie sie arbeitet. An diesen Stellen ist eine knappe, fachliche Beschreibung in Form eines oder mehrerer Inline-Kommentare erlaubt.

Besonders bei der Nutzung von SQL-Abfragen innerhalb des Quellcodes kann es hilfreich sein, diese in einem Kommentar kurz an der jeweiligen Stelle zu beschreiben.

Richtlinien zur Fehlerbehandlung

Abschließend soll das Dokument sich mit der Behandlung und Vermeidung von Fehlern im Quellcode befassen und Vorgaben machen, wie beispielsweise korrekte Werte oder sinnvolle Fehlermeldungen genutzt werden können. Ergänzend werden weitere Möglichkeiten vorgeschlagen, Fehler während der Entwicklung bereits zu entdecken und korrekt zu behandeln. Dabei wird sowohl auf die Überprüfung von Parametern und anderen Werten, sowie auf den richtigen Umgang mit Exceptions eingegangen.

6.1 Fehlerbehandlung im Code

Parameter werden nach Möglichkeit und Notwendigkeit innerhalb der Methode, in die sie übergeben werden, auf Gültigkeit und Korrektheit geprüft.
Dabei wird festgestellt, ob ein Parameter einen gültigen Wert besitzt und initialisiert ist. Weiterhin kann überprüft werden, ob der Wert eines Parameters in einem vorgegebenen Bereich liegt, oder aber einen exakten Wert besitzt. Dies hängt natürlich von der Benutzung innerhalb der Methode ab und sollte dementsprechend implementiert werden. Ein Beispiel für eine solche Überprüfung von Parametern kann wie in Quellcode 6.1 dargestellt aussehen:

```
        // 1. Parameter soll zwischen 1 und 10 liegen
        // 2. Parameter darf nicht leer sein
        public void TestMethod(int number, string text){
                if(number < 1 || number > 10 ||
                   ↪ String.IsNullOrEmpty(text)){
                        // Fehlerbehandlung...
                }
                // Eigentlicher Inhalt der Methode...
        }
```

Quellcode 6.1: Parameterpruefung in Methoden

6.2 Exceptions

Exceptions sollen für die Behandlung von Fehlern im Ablauf des Programms verwendet werden, um eine korrekte Fehlerbehandlung zu ermöglichen. Dazu ist es nötig, Exceptions an den passenden Stellen einzusetzen und passende Maßnahmen einzuleiten, mit diesen umzugehen. Sollte eine Exception in einem catch-Block erwartet werden, so ist dieser auch mindestens mit einer Ausgabe der Fehlermeldung oder mit einer direkten Behandlung der Ausnahme zu füllen.
Exceptions sind mit einem kleinen e zu benennen, dies sticht aus dem gewählten Schema für die Namen von Bezeichnern heraus und es ist sichergestellt, dass es sich um den Bezeichner einer Exception handelt.
In Quellcode 6.2 ist ein Beispiel einer einfachen Ausnahmebehandlung zu sehen.

```
try
{
        // Berechnung, Zuweisung, Aufruf usw.
}
catch (Exception e)
{
        // Fehlerbehandlung und Ausgabe der Meldung:
        SAPbouiCOM.Framework.Application.SBO_Application
        .StatusBar.SetText("Fehlermeldung: " +
          ↪ e.Message, BoMessageTime.bmt_Short,
          ↪ BoStatusBarMessageType.smt_Error);
}
```

Quellcode 6.2: Nutzung von Exceptions

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Definition von Richtlinien für die Entwicklung von SAP Business One Erweiterungen zur Verbesserung der Softwarequalität

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original