Masterarbeit

Optimierung der Softwarearchitektur und der Workflows am Beispiel von Anwendungs-Plug-ins und spezialisierten Frameworks

Andreas Brandhoff

geboren am 05. Februar 1988 in Paderborn

Studiengang Informatik Master

Westsächsische Hochschule Zwickau
Fakultät Physikalische Technik/Informatik
Fachgruppe Informatik

Betreuer, Einrichtung: Prof. Dr. W. Golubski,
WH Zwickau
Dipl.-Inf. (FH) A. Naumann,
N+P Informationssysteme GmbH

Abgabetermin: 03. Januar 2018

Autorenreferat

Häufig unterliegen historisch gewachsene Softwareprojekte nur schwach der technologischen Evolution. Modernisierungen und Optimierungen der Arbeitsweise und der Technologien werden nur selten oder langsam umgesetzt. Dabei kann der Pflegeaufwand für den bestehenden Code unbemerkt steigen und das Entwicklungsteam um seine Innovationskraft berauben.

Im Rahmen dieser Arbeit wird ein solches Softwareprojekt einer Generalüberholung unterzogen. Mit einem Continuous Delivery-Verfahren legt die Arbeit den Grundstein für automatisierte Prozesse innerhalb des Projektes. Darauf aufbauend wird zur Qualitätssteigerung ein System zur Automatisierung der Softwaretests etabliert. Ebenso werden Inspektionen zur Codequalität etabliert. Das Ende des Continuous Delivery-Prozesses stellt die automatisierte Auslieferung der Produkte dar. Hierzu wird ein Aktualisierungsverfahren entwickelt, dass die Versionierung von Teilkomponenten berücksichtigt und so das Ausliefern einzelner Komponenten ermöglicht.

Parallel zu den genannten Optimierungen wird die nach Bedarf gewachsene Architektur analysiert. Anschließend wird diese überarbeitet und moderne Technologien integriert. Im Rahmen dieser Modernisierung wird die Architektur auch einer Inspektion unterzogen und an allgemeine Richtlinien angepasst.

Durch die Modernisierung werden sowohl die Wartbarkeit als auch die Qualität des Projektes verbessert. Außerdem werden Richtlinien zur Qualitätssicherung während der Weiterentwicklung des Projektes etabliert um eine nachhaltigen Nutzen aus der Modernisierung zu ziehen.

Betreuende Einrichtung

Diese Arbeit wird in Zusammenarbeit mit der Firma N+P Informationssysteme GmbH in Meerane durchgeführt. Die Firma wurde 1990 in Meerane gegründet. Heute umfasst sie sieben Standorte mit 160 Mitarbeitern. Als Dienstleister bietet die N+P Informationssysteme GmbH IT-Lösungen für den Produkt- und Gebäudeentstehungsprozess. Besonderes Herausstellungsmerkmal der Firma ist die Systemintegration über die gesamte Wertschöpfungskette.

Betreuende Abteilung ist die auf den Produktentstehungsprozess spezialisierte Abteilung CAD-CAM-PDM(CCP). Neben verschiedenen Dienstleistungen entwickelt die CCP-Abteilung die NuPTools. Dies sind Plug-ins für die Anwendungen Inventor® und Vault von Autodesk®.

Inhaltsverzeichnis

Glossar

Autodesk® Inventor® Autodesk® Inventor® ist eine auf Modellierungselementen aufbauende, parametrische 3D-CAD-Software. Die Software wurde speziell für die mechanische Konstruktion konzipiert und findet insbesondere Verwendung in Maschinenbau, Werkzeugbau, Blechverarbeitung und Anlagenbau (Wikipedia 2017a).

Autodesk® Vault Autodesk® Vault ist ein Verwaltungswerkzeug für Konstruktionsdaten. Es unterstützt die Anwender bei der Zusammenarbeit und ermöglicht ein Revisionsmanagement. (Wikipedia 2016a).

Build Agent Ein TeamCity Build Agent ist eine Software die Befehle eines TeamCity Servers entgegennimmt und den eigentlichen Erstellungsprozess startet. Sie wird getrennt vom TeamCity-Server installiert und konfiguriert. Ein Agent kann auf dem selben Computer wie der Server oder einer anderen Maschine installiert sein (Megorskaya 2016).

Constructor Injection Contructor Injection ist eine Technik beim IoC-Muster, bei dem Abhängigkeiten über Kontruktorparameter eingebunden werden.

Domain-specific Language Eine domänenspezifische Sprache oder anwendungsspezifische Sprache ist eine formale Sprache, die zur Interaktion [...] für ein bestimmtes Problemfeld (die sogenannte Domäne) entworfen und implementiert wird (Wikipedia 2016b).

Feature Branch Feature Branches sind eine Arbeitsweise in SCM-Systemen. Sie werden genutzt um neue Funktionen für zukünftige Versionen zu entwickeln. [...] Das Wesen eines Feature Branches ist, dass er so lange existiert, wie die Funktion in Entwicklung ist und schlussendlich mit dem allgemeinen Branch zusammengeführt wird (Driessen 2015).

Hostanwendung Mit Hostanwendung wird die Anwendung bezeichnet, in die ein Plug-in hineingeladen wird. Im NuPTools-Projekt sind dies die Anwendungen Autodesk® Inventor® und Autodesk® Vault.

Intermediate Language Intermediate Language bzw. Common Intermediate Language ist eine Zwischensprache, in die alle Programme der Common Language Infrastructure übersetzt werden. Dies Umfasst unter Anderem das .Net-Framework (Wikipedia 2017b).

NuGet NuGet ist der Paketmanager für .Net (.NET Foundation 2017).

NuPTools NuPTools ist die Marke, unter der N+P Informationssysteme GmbH Plug-ins und Apps für Autodesk® Produkte entwickelt und vertreibt.

Prerelease Ein Prerelease ist ein Release, das zu Entwicklungs- oder Testzwecken bereitgestellt wird. Es kann i.d.R. nicht als stabil und fehlerfrei angesehen werden.

Release Ein Release ist eine bestimmte Version einer Software. Sie kann öffentlich oder intern bereitgestellt werden.

Repository Host Ein Repository Host ist ein Server, der zum Verwalten von Quellcodes verwendet wird. I.d.R stellen diese Server zusätzliche Funktionen wie ein Ticketsystem oder ein Wiki zur Verfügung.

Stakeholder Ein Stakeholder eines Systems ist eine Person oder Organisation, die (direkt oder indirekt) Einfluss auf die Anforderungen des betrachteten Systems hat (Pohl und Rupp 2015).

Abkürzungsverzeichnis

CD Continuous Delivery.

CDep Continuous Deployment.

CI Continuous Integration.

DSL Domain-specific Language.

IL Intermediate Language.

IoC Inversion of Control.

Nupis N+P Informationssysteme.

R# CLT ReSharper Command Line Tool.

SCM Source Control Management.

TFS Team Foundation Server.

WPF Windows Presentation Foundation.

Kapitel 1

Einleitung

Eine natürliche Evolution in der Softwareentwicklung sind Änderungen am Code- und Architekturstil. Es ist aber ebenso der vorherrschende Standard, dass Projekte eben diese Evolution scheuen und auf ihrem Stand stehen bleiben. Zu groß ist die Angst vor dem Änderungsaufwand an bestehendem Code. Dabei werden häufig die Vorteile der Neuerungen außer acht gelassen. Der Änderungsaufwand wird durch die Optimierungen in der Arbeitsweise und der Pflege des Altbestandes schnell ausgeglichen. Besonders auf Managementebene wird dies schnell übersehen.

In der Vergangenheit wurde Code händisch von einem Entwickler kompiliert. Heute wird dies von Continuous Delivery-Systemen übernommen. Diese sind das zentrales Werkzeug eines modernen Softwareprojektes. Als autonom arbeitende Systeme sind sie die Basis für Automatisierungen. Neben dem Erstellen des Projektes werden daher häufig auch qualitätssichernde Prozesse und sogar Auslieferungsverfahren an diese Systeme übertragen.

Qualitätssicherung in der Softwareentwicklung setzt an verschiedenen Punkten an. Offensichtlichstes Qualitätsmerkmal ist die Funktionalität. Diese kann durch automatisierte Tests sichergestellt werden. Mit Hilfe verschiedener Werkzeugen kann heutzutage fast jeder Aspekt einer Software getestet werden. Durch die Automatisierung können Funktionalitätstests nahezu kostenfrei und zu jedem Zeitpunkt wiederholt werden.

Ein weiteres Qualitätsmerkmal ist der Codestil. Der Codestil wirkt sich praktisch unmittelbar auf die Pflege des Codebestandes aus. Übersichtlicher Code, der vorgegebenen Konventionen entspricht, ist für den geübten Leser leichter zu verstehen. Das Verständnis eines bestehenden Codes ist eine erforderliche Voraussetzung für dessen Bearbeitung. Somit beeinflusst der Codestil die Entwicklungszeit stark.

Die Variationen in Auslieferungsverfahren von Softwareprojekten umfassen verschiedenste Möglichkeiten, von vollständig händisch bis praktisch unsichtbar für den Anwender. Der Grad der nötigen Automatisierung ist von verschiedenen Aspekten wie beispielsweise der Anwenderanzahl oder etablierten Routinen abhängig. Besonders bei langlebigen Projekten können sich diese Aspekte ändern. Die Anforderungen an das Auslieferungsverfahren können sich daher über die Zeit deutlich ändern. Für Unternehmen entsteht hier eine besondere Hemmschwelle. Da bereits ein Verfahren existiert, sieht besonders die Managementebene keinen Verbesserungsbedarf.

Die Architektur einer Software definiert die Komponenten und deren Interaktion. Sie hat damit einen großen Einfluss auf die Qualität des Projektes. Viele Prinzipien und Richtlinien beschreiben Eigenschaften von guten Architekturen. Verschiedene Themen von der Größe und dem Umfang von Komponenten bis zu deren Anordnung und Anwendung werden hiervon abgedeckt. Vielen Entwicklern sind diese Empfehlungen jedoch nicht bekannt oder werden nicht befolgt. Besonders Quereinsteigern in die Softwareentwicklung fehlt das Wissen über solche Vorgaben. Verstöße gegen die Prinzipien und Richtlinien kommen insbesondere bei der Pflege der Software zum tragen. Eine klare Ausarbeitung der Architektur wirkt sich daher positiv auf den Pflegeaufwand eines Softwareprojektes aus.

Besonders in langlebige Projekte sind daher verschiedenste Sanierungsmöglichkeiten zu finden. Das NuPTools-Projekt der Firma N+P Informationssysteme GmbH ist ein Vertreter dieser historisch gewachsenen Projekte. Es umfasst z.Z. ca. 50 kleine bis mittlere Plug-ins für die Anwendungen Autodesk® Inventor® und Autodesk® Vault. Aufgrund des Umfangs und des aktuellen Zustandes ist die Weiterentwicklung des Projektes nahezu unmöglich. Das NuPTools-Projekt wird beispielhaft runderneuert. Dabei wird es zu einem modernen und flexiblen System umgebaut um eine einfache und effiziente Weiterentwicklung wieder zu ermöglichen.

Kapitel 2

Continuous Delivery

Continuous Delivery (CD) bezeichnet ein Verfahren zum automatisierten Erstellen von Software. Der Funktionsumfang eines CD-Systems umfasst neben dem simplen Erstellen der Software beispielsweise auch Aufgaben wie das Testen oder Bereitstellen von Paketen.

2.1 Gründe und Vorteile von Continuous Delivery

Wolff (2014) nennt zur Einführung in das Thema verschiedene Gründe und Vorteile für ein CD-System. Insbesondere wird dabei auf die Geschwindigkeit der Rückmeldungen eingegangen. Während sich der Entwickler auf die aktuellen Komponenten konzentrieren kann, erstellt das CD-System das gesamte Projekt und testet es. Dieser Vorgang läuft automatisiert ab und erzeugt keinen Aufwand bei dem Entwickler. Dennoch erhält dieser praktisch umgehend Rückmeldungen von dem System und den Ergebnissen. Durch diese automatisierte Integration können geänderter Komponenten häufiger und schneller integriert werden. Probleme bei der Interaktion zwischen verschiedenen Komponenten fallen daher umgehend auf und werden aufgrund der kleineren Menge an Änderungen handhabbarer.

Auch die Bereitstellung für Tester oder Kunden kann sofort erfolgen. Im Gegensatz zu klassischen Auslieferungsverfahren, bei denen neue Versionen oft mit mehren Wochen oder Monaten Abstand erstellt werden, verkürzt sich hier die Zeit zwischen der eigentlichen Entwicklung und dem Feedback der Nutzer enorm.

Eine weiterer Eigenschaft sind die automatisierten Tests. Das Ausführen der Tests erfordert keine Aufmerksamkeit eines Mitarbeiters. Es empfiehlt sich daher die Automatisie-

rung möglichst vieler Tests. Dies ermöglicht das häufige Wiederholen von Tests. Fehler durch geänderte Komponenten fallen somit umgehend auf.

Auch die Testumgebungen können automatisiert werden. Durch das automatisierte Erstellen der Testumgebungen sind die Einflüsse auf die Tests besser kontrollierbar. Außerdem sind Fehler leichter Reproduzierbar, da unbekannte Einflüsse, die beispielsweise auf einem Entwicklerrechner bestehen würden, minimiert werden.

Ein weiterer Vorteil der so durchgeführten Tests im Vergleich zu händisch durchgeführten Tests ist, neben dem geringeren Aufwand, die geringere Fehlerquote bei der Testdurchführung. Dies wird durch die Häufigkeit der Testdurchführung und der damit verbundenen geringeren Änderungsmenge begründet.

Bei Continuous Delivery-Systemen werden die Versionen aller erstellten Komponenten auf einem zentralen System gelagert. Hierdurch wird nicht nur die Gefahr durch menschliche Fehlorganisation eliminiert. Das System kann auch jederzeit auf eine ältere Version zurück wechseln, falls beispielsweise Fehler bemerkt werden oder alte Versionsstände untersucht werden sollen. Dies kann bei Problemen dieser Art deutliche Verbesserungen im Entwicklungsprozess bringen.

Zusammenfassend kann man drei wesentliche Vorteile von Continuous Delivery Systemen nennen. Diese sind das schnelle Feedback, wodurch die erneute Einarbeitungszeit des Entwicklers entfällt. Die höhere Qualität durch die deutlich gesteigerte Testanzahl. Und die automatisierte Versions- bzw. Release-Verwaltung.

2.2 Ablauf von Continuous Delivery

Die Aufgabe eines Continuous Delivery-Systems ist die Abarbeitung von Schritten zum Erstellen, Testen und Ausliefern einer Software. Diese im Allgemeinen als Pipeline oder Chain bezeichnete Abfolge wird dem jeweiligen Bedarf angepasst. Dennoch sind einige generelle Stufen üblich. Dies zeigen beispielsweise Wolff (2014) und Duvall, Matyas und Glover (2009).

2.2.1 Stufe 1: Integration

Die erste Stufe in der Pipeline ist in der Regel die Integration. In diesem Schritt werden die Komponenten zusammengefügt und erstellt. Im Detail bedeutet dies, dass die aktuelle Version von einem Source Control Management (SCM)-Server geladen wird. Außerdem

müssen verwendete Abhängigkeiten wie bereits erstellte Komponenten oder Fremdkomponenten bereitgestellt werden. Idealerweise steht hierfür ein Paketmanagementsystem wie NuGet[1] oder npm[2] zur Verfügung. Neben dem produktiven Code können in dieser Stufe auch zusätzliche Komponenten wie beispielsweise die Komponententests erstellt werden.

2.2.2 Stufe 2: Test

Die nächste Stufe der Build-Pipeline umfasst verschiedene Formen von Tests. Hier tritt ein Vorteil eines CD-Servers besonders deutlich hervor. Da diese Tests bis auf einige Ausnahmen vollständig automatisiert sein sollten, können ausführliche Tests ohne die Aufmerksamkeit eines Entwicklers erfolgen.

Neben Komponententests, welche die Funktionalität der entwickelten Software überprüft, sind Deployment-Tests eine der wichtigsten Testformen. Beim Deployment wird die Software in einer Testumgebung installiert und auf Ihre Lauffähigkeit getestet. Durch die Automatisierung besteht hier die Möglichkeit verschiedene Systeme zu testen.

Auf die Arten und Möglichkeiten der Testautomatisierung wird in Kapitel 3 auf Seite 19 detaillierter eingegangen.

2.2.3 Stufe 3: Inspektion

In der dritten Stufe werden häufig Code-Inspektionen implementiert. Hierzu stehen je nach verwendeter Programmiersprache verschiedene Tools zur Überprüfung zur Verfügung. Beispielsweise führen im .Net Bereich Tools wie StyleCop[3] oder ReSharper Command Line Tools[4] eine Analyse des Codestils durch. Hierbei können neben den üblichen Konventionen wie Schreibweisen beispielsweise auch fehlende Codedokumentationen erfasst werden.

2.2.4 Stufe 4: Auslieferung

Die letzte Stufe ist die Auslieferung. Hier wird die erstellte Software für die Verwendung zur Verfügung gestellt. Dies kann je nach Anforderung auf verschiedene Weisen erfolgen

[1] https://www.nuget.org/
[2] https://www.npmjs.com/
[3] https://github.com/StyleCop
[4] https://www.jetbrains.com/resharper/features/command-line.html

und ist nicht auf das schlichte Anbieten der erstellten DLL-Dateien beschränkt. Automatisiert können Pakete für zuvor erwähnte Paketmanager erstellt und veröffentlicht werden oder eigene Update-Verfahren integriert werden.

Ebenfalls ist die automatisierte Installation in das Produktivsystem möglich. Dies findet häufig bei selbstbetriebenen Systemen wie Websites Anwendung.

2.3 Differenzierung der Begriffe

In der Continuous Delivery Thematik sind häufig drei Unterschiedliche Begriffe zu finden. Diese sind Continuous Integration (CI), Continuous Delivery (CD) und Continuous Deployment (CDep). Zwar beschreibt jeder dieser Begriffe eine Build-Pipeline, dennoch bestehen Unterschiede zwischen den Begriffen. Fowler (2013) geht auf die verschiedenen Ausprägungen ein und differenziert die Begriffe.

Die Bezeichnung Continuous Integration beschreibt die in Abbildung 2.1 gezeigten Schritte. Das Ergebnis dieser Schritte ist das automatisiert erstellte und getestete Programm.

Die nächst größere Automatisierungsstufe wird von dem Begriff Continuous Delivery beschrieben. Diese Variante ist in Abbildung 2.2 auf der nächsten Seite zu sehen. Hier folgen auf die als Continuous Integration beschriebenen Schritte zusätzlich der Schritt Deployment sowie der namensgebenden Schritt Delivery. Bei dem Deployment Schritt wird das frisch erstellte Programm in einer Testumgebung installiert. Hierdurch wird sichergestellt, dass die neuen Änderungen keine Probleme bei der Installation oder Laufzeit erzeugen. Nach dem erfolgreichen Deployment Test befindet sich die erstellte Version in einem „Ready-For-Delivery“-Zustand. Nun kann ein Entwickler manuell die Auslieferung bzw. Bereitstellung auslösen.

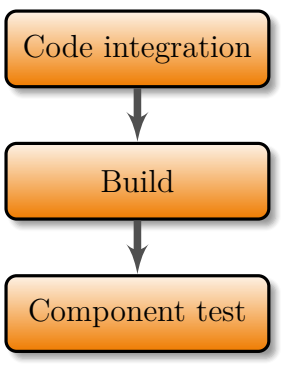

Abbildung 2.1: Schritte eines Continuous Integration-Systems

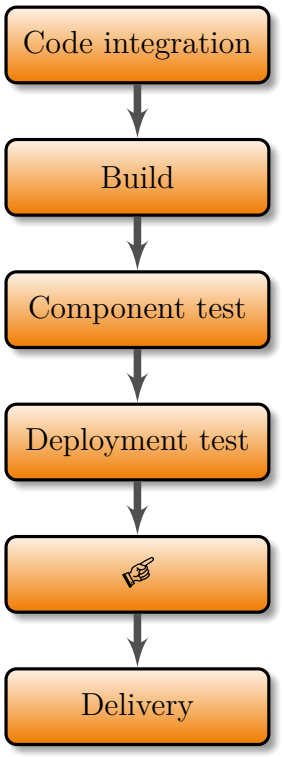

Abbildung 2.2: Schritte eines Continuous Delivery-Systems

Der dritte der Begriffe, Continuous Deployment, ist aufgrund der üblicherweise gleichen Abkürzung leicht mit Continuous Delivery zu verwechseln. Wie in Abbildung 2.3 auf der nächsten Seite zu sehen ähneln sich auch diese Schritte sehr. Im Gegensatz zu dem vorangegangen Verfahren wird die Auslieferung jedoch ohne Entscheidung eines Entwicklers ausgelöst. Jede Änderung, die erfolgreich die Pipeline durchquert hat, resultiert in eine Aktualisierung der Produktivsysteme. Wie bei der Deployment Phase auch ist hier die Bezeichnung aus der Webentwicklung namensgebend. Als Deployment wird hier das Installieren und in Betrieb nehmen der neuen Software bezeichnet.

Der maßgebliche Unterschied zwischen Continuous Delivery und Continuous Deployment ist also die Kontrolle und Geschwindigkeit der Auslieferung der Änderungen. In der Continous Deployment-Thematik wird häufig das Beispiel einer Website genannt. Hier kann ein vollständig automatisiertes Continuous Deployment-System Änderungen beispielsweise in einem Canary-Verfahren bis in das Produktivsystem liefern.

Bei dem Canary-Verfahren (Sato 2014) werden neue Produktversionen auf einem zweiten Server installiert. Die Besucher der Website werden nach einer definierten Aufteilung auf das alte oder neue System weitergeleitet. So werden nur kleine Mengen der Besucher auf das neue System geleitet. Stufenweise wird die Aufteilung zugunsten des neuen Systems

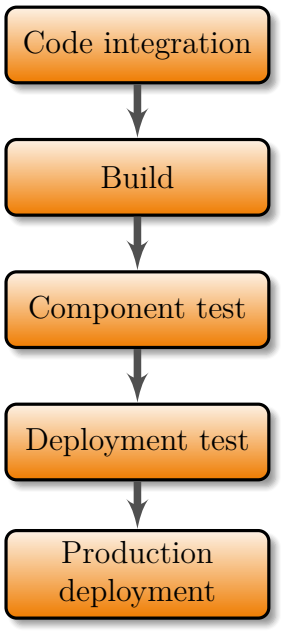

Abbildung 2.3: Schritte eines Continuous Deployment-Systems

geändert, bis dieses das Altsystem vollständig abgelöst hat.

Dieser Prozess kann bei einigen Projekten mehrmals täglich beginnen oder mit mehreren Versionen parallel erfolgen. Dies ist jedoch nicht in jedem Bereich erwünscht. Beispielsweise ist es bei Desktop Anwendungen nicht üblich mehrere Updates pro Tag anzubieten. Hier bietet das Continuous Delivery Verfahren durch ein manuelles oder Zeitplan gesteuertes Ausliefern die Möglichkeit, die Geschwindigkeit auf das gewünschte Maß zu drosseln.

2.4 Ziele

Die Firma N+P Informationssysteme hat mit ihren Produkten der NuPTools-Produktgruppe sehr spezielle Anforderungen an den Erstellungsprozess. Daher soll ein Continuous Delivery-Verfahren den Prozess verbessern. Ziel ist es die Mitarbeiter zu entlasten sowie die Qualität der Produkte zu erhöhen.

Da die NuPTools eine große Menge Einzelkomponenten umfassen erfordert die Erstellung der gesamten Produktgruppe viel Zeit. Neue Versionen werden daher nur selten veröffentlicht. Häufig umfassen sie nur dringende Fehlerkorrekturen. Aufgrund der Updatestrategie der Hostanwendungen muss jedoch jedes NuPTool jährlich aktualisiert und neu erstellt werden. Hier soll durch die automatisierte Erstellung Abhilfe geschaffen werden. Durch diese Entlastung soll zusätzlich ein schnellerer Updaterhythmus ermöglicht werden um

den Anwendern sowohl Fehlerkorrekturen als auch Verbesserungen schneller verfügbar zu machen.

Ein besonderer Augenmerk liegt auf der Testautomatisierung. Zur Zeit können die NuP-Tools-Produkte nur händisch geprüft werden. Neben dem erheblichen Zeit- und Personalaufwand für diese Tests sind diese auch sehr Lückenhaft. Häufig werden nicht alle Funktionalitäten untersucht oder Anforderungen an die Testumgebung nicht wahrgenommen, da diese „zufällig“ auf dem Testsystem gegeben waren. Auch sind in der Vergangenheit Probleme bei Kombinationen bestimmter Einzelkomponenten aufgetreten. In der Zukunft sollen daher systematisierte Deployment-Tests durchgeführt werden.

Zusätzlich zu den bereits genannten Zielen soll durch das System ein zentraler Punkt zum Erstellen und Ausliefern von NuPTools-Produkten geschaffen werden. In der Vergangenheit sind vereinzelt Probleme in der Versionierung aufgetreten. Entwickler haben aktuelle Versionen lokal erstellt. Die Versionen wurden jedoch nicht ordnungsgemäß in der bisherigen Versionsverwaltung archiviert haben. Hierdurch wurde die Reproduzierbarkeit von Fehlern erschwert oder unmöglich gemacht. Durch einen zentralen Server der jede auszuliefernde Version erstellt und archiviert soll dieses Problem gelöst werden. Auch die Kernkomponenten, die in vielen Produkten zum Einsatz kommen, sollen an einer zentralen Stelle erstellt und verwaltet werden.

2.5 Systemauswahl

Auf den ersten Blick stehen viele verschiedene Anbieter für Continuous Delivery-Systeme zur Verfügung. Viele dieser Anbieter versuchen durch Alleinstellungsmerkmale herauszustechen. So reicht die Auswahl von extrem einfach zu konfigurierenden Systeme bis zu stark anpassbaren Systeme. Leider konzentrieren sich viele dieser Anbieter stark auf Cloud-Lösungen. Daher werden viele Systeme nur als Service angeboten oder sind von Hostingplattformen wie GitHub[5] abhängig. Im Folgenden sind die untersuchten Systeme beschrieben.

[5] https://github.com/

2.5.1 CircleCI

CircleCi[6] ist ein System das sich besonders durch die einfache und schnelle Konfiguration auszeichnet. Zur Konfiguration wird der Zugriff auf einen GitHub oder GitHub Enterprise[7] Account erlaubt. Im zweiten Schritt wird das gewünschte Projekt ausgewählt und erstellt. Das Projekt wird daraufhin automatisch konfiguriert und die benötigten Schritte erkannt. Dies ist das vom Konfigurationsaufwand einfachste untersuchte System. Allerdings wird das .Net-Framework offiziell nicht unterstützt.

2.5.2 Teamfoundation Server

Der Team Foundation Server (TFS) von Microsoft bietet neben den Source Control Management-Funktionen auch Continuous Delivery-Funktionalitäten. Jedoch ist das System von der Bedienbarkeit her nicht Benutzerfreundlich. Durch die Komplexität der Konfiguration ist es einem unerfahrenen Benutzer praktisch unmöglich das System zu konfigurieren.

2.5.3 Jenkins

Eines der bekanntesten Systeme auf dem Markt ist Jenkins[8]. Das OpenSource Projekt zeichnet sich besonders durch seine Anpassbarkeit aus. Es steht für jede moderne Plattform zur Verfügung. Durch die große Community um Jenkins sind viele Informationen und Hilfestellungen zur Konfiguration verfügbar. Lediglich auf einen individuellen Support muss verzichtet werden. Das Jenkins-Projekt steht unter einer OpenSource-Lizenz und ist vollständig kostenfrei nutzbar. Die .Net-Integration lässt sich problemlos mit einigen Plug-ins nachinstallieren und umsetzen.

2.5.4 GitLab

Gitlab[9] ist, wie der TFS von Microsoft auch, vom Kern her ein Repository Host. Das System bietet jedoch einen großen Umfang an zusätzliche Funktionen. Darunter auch Continuous Delivery. GitLab verzichtet hier größtenteils auf Konfigurationsoberflächen. Statt dessen werden die Pipelines in den Projekten über ein eigenes Skript konfiguriert.

[6] https://circleci.com
[7] https://enterprise.github.com/home
[8] https://jenkins.io/
[9] https://about.gitlab.com/

Dieses Skript befindet sich zusammen mit dem Projekt in der Versionsverwaltung. Die CD-Konfiguration ist daher immer an einen Versionsstand des Projektes gebunden. Änderungen am Projekt, die Änderungen an der CD-Konfiguration nach sich ziehen oder umgekehrt, erzeugen daher keine Probleme beim erneuten Erstellen alter Versionsstände.

2.5.5 TeamCity

Das TeamCity[10]-System von Jetbrains ist besonders auf den .Net-Markt spezialisiert. Es unterstützt nativ .Net-Projekte. Die Konfiguration wird durch eine intelligenten Erkennung von Schritten unterstützt. Weitere auf .Net-Projekte ausgelegte Features wie beispielsweise eine NuGet-Server oder die Installation von .Net-Testwerkzeugen sind bereits integriert. Ebenso wie Jenkins steht es für jede Plattform zur Verfügung und ist mit wenig Aufwand einsatzbereit. Allerdings steht das kommerzielle Produkt nur eingeschränkt kostenlos zur Verfügung.

2.5.6 Fazit

Das CircleCI-System ist aufgrund der fehlenden .Net-Unterstützung keine Option. Auch der Team Foundation Server fällt nicht in die engere Wahl. Ein TFS ist bereits zum Source Code Management im Betrieb. Aufgrund der Komplexität scheidet er jedoch als Continuous Delivery-System aus. Die drei letztgenannten Systeme, GitLab, Jenkins und TeamCity, können die Anforderungen an den Erstellungsprozess erfüllen. Im direkten Vergleich sticht jedoch die bessere .Net-Unterstützung des TeamCity Servers hervor. Zusätzlich fließt die Konfiguration über die benutzerfreundliche Oberfläche des TeamCity Systems positiv in die Entscheidung ein. Daher wird für den Build-Prozess ein TeamCity-System konfiguriert.

2.6 Systemanpassungen

Zur Vorbereitung auf die Produkterstellung mittels Continuous Delivery-Systems werden Anpassungen an der Dateistruktur des NuPTools-Projekt vorgenommen. Außerdem werden Änderungen an der Dateiversionsverwaltung gemacht.

[10] https://www.jetbrains.com/teamcity/

2.6.1 Änderung der Versionsnummerierung

Die Versionsnummerierung der Komponenten und Produkte wird bisher sehr einfach gehalten indem eine zentrale Versionsdefinition in jedes Projekt eingebunden wird. Dadurch kann die Version für alle Komponenten gemeinsam an einer Stelle konfiguriert werden. Durch die Automatisierung des Erstellungsprozesses bietet sich die Möglichkeit eine funktionalere Variante umzusetzen.

Gregsdennis (2016) beschreibt die Problematik und die Möglichkeiten der Versionierung von .Net-Assemblies. Das Versionsnummernschema einer .Net-Anwendung besteht aus vier durch Punkte getrennte ganzzahlige Werte. Die erste Stelle wird Hauptversionsnummer oder Major Release genannt. Änderungen an dieser Stelle indizieren große Änderungen. Gleichzeitig indiziert eine gleichbleibende Hauptversion eine Aufwärtskompatibilität innerhalb der Hauptversion. Es sollte also sichergestellt sein, dass eine Anwendung die mit einer Abhängigkeit in der Version 2.1 erstellt wurde auch mit einer Version 2.4 lauffähig ist, jedoch nicht zwangsweise mit Version 3.0.

Die zweite Stelle des Schemas wird Nebenversionsnummer oder Minor Release genannt. Änderungen an dieser Stelle deuten Änderungen oder Erweiterungen an. Diese beeinflussen die Funktionalität jedoch nicht wesentlich.

Die Bedeutung der beiden weiteren Stellen unterscheidet sich je nach Bedarf wie ein Vergleich mit Cardella (2008) zeigt. Gemein haben die Bedeutungen in der Regel jedoch, dass es sich dabei um Informationen aus dem Erstellungsprozess handelt. Dies könnten beispielsweise ein kodiertes Datum oder eine Revisionsnummer aus dem Source Control Management (SCM)-System sein.

Die NuPTools verwenden das in Ausschnitt 2.1 gezeigte Schema. Dieses beinhaltet für die ersten beiden Stellen die Haupt- und Nebenversionsnummern. Die dritte Stelle ist nicht definiert und wird für zukünftige Änderungen reserviert. Auf der letzten Stelle wird die Buildnummer des CD-Systems verwendet. Dies ist eine Laufnummer, die bei jedem Erstellungsprozess erhöht wird.

```
Hauptversion.Nebenversion.0.Buildnummer
```

Ausschnitt 2.1: Versionsnummernschema der NuPTools

.Net bietet durch die Möglichkeit verschiedene Versionsnummern zu definieren eine hohe Flexibilität. Mit der in Ausschnitt 2.2 auf der nächsten Seite gezeigten Eigenschaft *AssemblyVersion* wird die technisch verwendete Versionsnummer festgelegt. Diese wird von

```
[assembly: AssemblyVersion("2.0.0.0")]
[assembly: AssemblyFileVersion("2.4.0.143")]
[assembly: AssemblyInformationalVersion("2.4.0.143-Beta")]
```

Ausschnitt 2.2: Versionsnummern in C♯

erstellten Programmen überprüft. Die Kompatibilität zwischen verschiedenen Nebenversionen wird erreicht indem hier nur die Hauptversion angegeben wird.

AssemblyFileVersion beschreibt die Dateiversion. Hier werden auch die Nebenversionsnummer und die Buildnummer gesetzt. Dadurch wird sowohl der funktionalen Stand die Datei beschrieben, als auch eine eindeutige Nummer durch die Buildnummer vergeben.

Die dritte Versionsnummer, die *AssemblyInformationalVersion*, wird zur Erstellung der NuGet-Pakete verwendet. Hier kann das Schema um einen beliebigen Tag erweitert werden. Beispiele für einen solchen Tag sind „Debug“ oder „Beta“.

2.6.2 Verwendung der eigenen NuGet-Pakete

Die größte Anpassung ist die vollständige Umstellung auf NuGet-Referenzen. Für die in verschiedenen NuPTools gemeinsam verwendeten Komponenten aus dem Unterprojekt NuPCore sind bereits NuGet-Pakete verfügbar. Allerdings wurden die NuPCore-Komponenten untereinander nicht über NuGet-Pakete referenziert. Jede NuPCore-Komponente liefert daher die von ihr verwendeten Komponenten mit. Dies ist jedoch nicht gewünscht, da Komponenten dadurch mehrfach bereitgestellt werden. Des Weiteren kann dies zu Versionskonflikten führen wenn Komponenten anstelle der aktuellen NuGet-Pakete eigene Assemblies verwenden.

2.6.3 Bereitstellung von Fremdkomponenten via NuGet

Für das Erstellen müssen verwendete Fremdkomponenten bereitgestellt werden. Es bieten jedoch nicht alle Anbieter ihre Komponenten als NuGet-Paket an. Um dennoch eine einheitliche und versionierte Verwaltung der Abhängigkeiten zu ermöglichen, werden alle Fremdkomponenten auf einem firmeninternen NuGet-Server veröffentlicht, sofern sie vom Anbieter nicht bereits angeboten werden. Dies löst die Abhängigkeit zu vorinstallierten Komponenten und Frameworks. Besonders auf dem Build-Server macht sich dies bemerkbar indem weniger Software vorinstalliert werden muss.

2.6.4 Anpassung der Dateistruktur

Eine weitere Anpassung ist die Überarbeitung der bereits existierenden Komponententests. Die vereinzelten Tests sind nicht einheitlich benannt oder positioniert. Um die Arbeit mit Templates auf dem Continuous Delivery-Server zu vereinfachen wird zu jeder Komponente ein Testprojekt mit gleichem Namen und dem Suffix „.Tests“ erstellt. Außerdem werden alle Komponententests in einem separaten Ordner abgelegt, um eine übersichtlichere Struktur zu erhalten.

2.7 Umsetzung mit TeamCity

Zur Umsetzung des Continuous Delivery-Verfahrens im NuPTools-Projekt wird ein TeamCity-Server eingesetzt. Dieser wird vollständig über eine grafische Oberfläche konfiguriert. Zusätzlich bietet das System viele benötigte Komponenten und Funktionen. Es müssen nur wenige Erweiterungen an dem System in Form von Plug-ins erfolgen. Zu den vorhandenen Funktionen gehören unter Anderem die NuGet-Unterstützung samt integriertem NuGet-Server, die NUnit-Unterstützung sowie verschiedene Möglichkeiten zur Inspektion des Codestils.

2.7.1 Grundlagen des TeamCity-Systems

In dem TeamCity-System werden die einzelnen Projekte als Build-Konfigurationen bezeichnet. Bei Bedarf können diese Konfigurationen zu einer Kette verbunden werden. Mit den Konfigurationen werden die einzelnen Arbeitsschritte und Auslöser definiert. Ausgeführt werden die Erstellungsprozesse auf Build Agents. Diese können entsprechend dem Bedarf unterschiedlich konfiguriert werden.

Als Auslöser können neben Einträge in einem verbundenen SCM-System und zeitlichen Bedingungen auch Abhängigkeiten dienen. So kann das Erstellen einer Konfiguration beispielsweise bei der Veröffentlichung eines NuGet-Paketes oder der Fertigstellung einer anderen Konfiguration ausgelöst werden.

Build-Konfigurationen setzen sich aus einzelnen Arbeitsschritte zusammen. TeamCity bietet hierbei verschiedene spezialisierte Schritte an. Mit diesen können die Konfigurationn einfach und intuitiv erstellt werden. Die in einer Build-Konfiguration zusammengefassten Schritte werden immer zusammen ausgeführt. Durch die Verkettung mehrere Konfigurationen können komplexere Abläufe definiert werden. Damit ist es beispielsweise möglich,

mehrere Konfigurationen parallel auf verschiedenen Build Agents auszuführen oder zur Testdurchführung einen speziell konfigurierten Agent zu verwenden. Durch intelligente Abhängigkeitsbeziehungen ist das System in der Lage zu erkennen, ob ein vorangehendes Kettenglied ausgeführt werden muss, oder beispielsweise schon von einer anderen Build-Konfiguration ausgelöst wurde.

Das TeamCity System bietet mehrere Komfortfunktionen. Diese vereinfachen sowohl die Einrichtung als auch die Pflege. Das NuPTools Projekt macht insbesondere von Templates und Parametervererbungen Gebrauch. Bei den Templates handelt es sich um generische Konfigurationen. Mit Hilfe von Parametern können diese spezialisiert werden. Die Instanzen der Templates bleiben dabei von dem Template abhängig. Änderungen an dem Template werden daher in allen Instanzen angewandt. Zusätzlich können die Instanzen Eigenschaften der Templates überschreiben und so weiter an den spezifische Bedarf angepasst werden.

Die Konfigurationen können in einer Dateiordnerähnlichen Struktur angeordnet werden. Dies bietet neben Übersichtlichkeit auch eine Parametervererbung an. Diese Vererbung ermöglicht es Parameter auf einer hohen Ebene zu definieren. Dadurch stehen sie ebenfalls auf niedrigeren Ebenen zur Verfügung stehen. Die Parameter eines übergeordneten Projektes werden somit in jeder Konfiguration angewendet.

2.7.2 Konfigurationsübersicht

Zur Einrichtung des NuPTools-Projektes werden mehrere eigene Templates verwendet. Abbildung 2.4 auf der nächsten Seite zeigt diese Konfigurationen exemplarisch. Für jede Komponente werden Instanzen dieser Templates erstellt. Die Konfigurationen „Build“, „Deploy“ und „Deliver“ werden für die Erstellung der Releases verwendet. Wie in der Abbildung zu sehen sind die Konfigurationen voneinander abhängig. Dies stellt sicher, dass eine Komponente erst veröffentlicht wird wenn sie alle Tests erfolgreich passiert hat.

Zu Entwicklungszwecken werden Prereleases erstellt. Diese werden ausschließlich zur Entwicklung verwendet. Um den Erstellungsprozess zu beschleunigen verzichten sie daher auf einige Schritte. Zusätzlich wird für jede Projektmappe eine Instanz des „Project Inspection“-Templates erstellt.

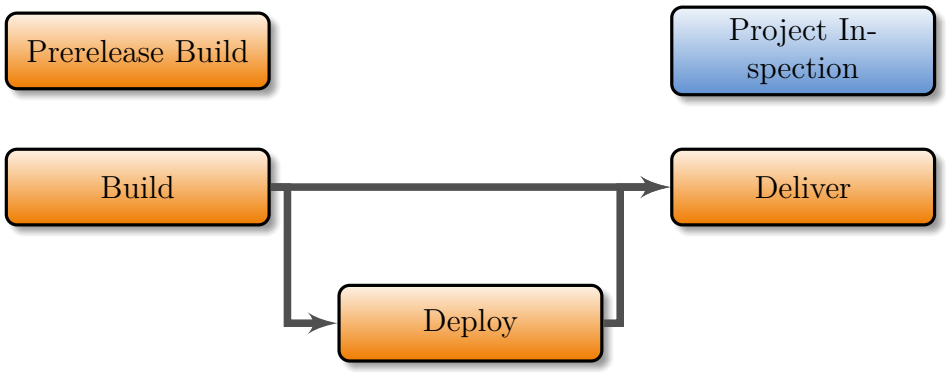

Abbildung 2.4: Exemplarische Anwendung der Build Templates

2.7.3 Prerelease-Konfiguration

Die in Abbildung 2.5 gezeigte Prerelease-Konfiguration dient zum Erstellen und Testen der Komponenten während des Entwicklungsprozesses. Sie wird über Einträge im SCM-System ausgelöst. Dabei werden auch Feature Branches berücksichtigt und Pakete mit einem entsprechenden Suffix erstellt.

Der Prozess beginnt mit der Wiederherstellung der verwendeten NuGet-Pakete. Dies ist nötig, da die Pakete nicht über das SCM verwaltet werden. Darauf folgend wird die Komponente sowie die dazugehörigen Tests erstellt.

Im nächsten Schritt wird die Erste von drei Testkategorien getestet. Diese Kategorie umfasst schnell laufende Tests. Anschließend wird die erstellte Komponente digital signiert. Die teilgetestet und signierte Komponente wird nun als NuGet-Paket firmenintern veröffentlicht.

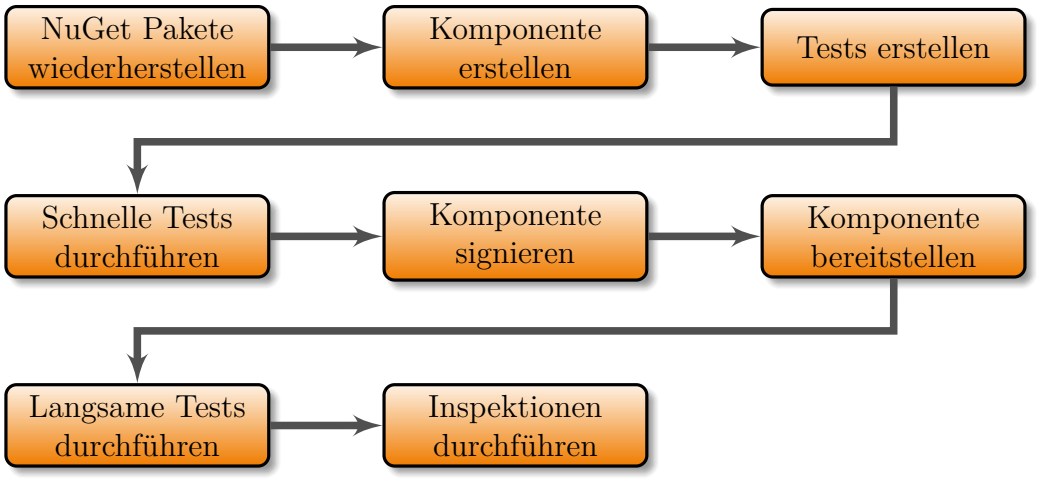

Abbildung 2.5: Ablauf der Komponentenerstellung zu Entwicklungszwecken

Die folgenden Schritte sind zeitaufwändiger. Sie werden daher erst nach der Bereitstellung der Komponente durchgeführt. Somit besteht die Gefahr, dass der Entwickler die Komponente nach der Paketerstellung noch nachbessern muss. Der Vorteil dabei ist jedoch, dass der Entwickler nicht lange auf die Komponente warten muss. Die nachfolgenden Tests der langsamen Kategorie umfassen Tests, die im Erfolgs- oder Fehlerfall mehrere Sekunden bis Minuten zur Durchführung benötigen. Dies kann beispielsweise bei Zeitüberschreitungen von Verbindungen der Fall sein. Zum Abschluss wird eine Codestilinspektion durchgeführt. Da diese kein Fehlverhalten der Software nachweist, stellt sie keinen Abbruchfaktor für den Erstellungsprozess dar.

2.7.4 Build-Konfiguration

Das erste Kettenglied bei der Erstellung und Veröffentlichung der Release-Versionen ist eine abgewandelte Version der in Abschnitt 2.7.3 auf Seite 16 beschriebenen Konfiguration. Sie wird ebenfalls über Einträge im SCM-System ausgelöst. In diesem Fall wird jedoch nur auf Branches reagiert, die der Veröffentlichung oder der Vorbereitung einer Veröffentlichung dienen.

Wie in Abbildung 2.6 zu sehen fehlen die Schritte „Komponente bereitstellen“ und „Inspektionen durchführen“. Das fehlen der Bereitstellung an dieser Stelle ist durch die in einer weiteren Konfiguration durchgeführten Tests begründet. Zu Entwicklungszwecken ist die vorübergehende Bereitstellung fehlerhafter Komponenten noch akzeptabel. Bei der Release-Veröffentlichung sollte dies jedoch vermieden werden. Die Paketerstellung wird daher erst nach Erfolg aller Tests durchgeführt. Da die Inspektion keine Fehler aufdeckt, sondern lediglich der Wartbarkeit des Codes zuträglich ist und die verwendeten Codeversionen ebenfalls auf dem fortlaufenden Entwicklungsbranch einer Inspektion unterzogen werden, wird bei der Release-Veröffentlichung auf die Inspektion verzichtet.

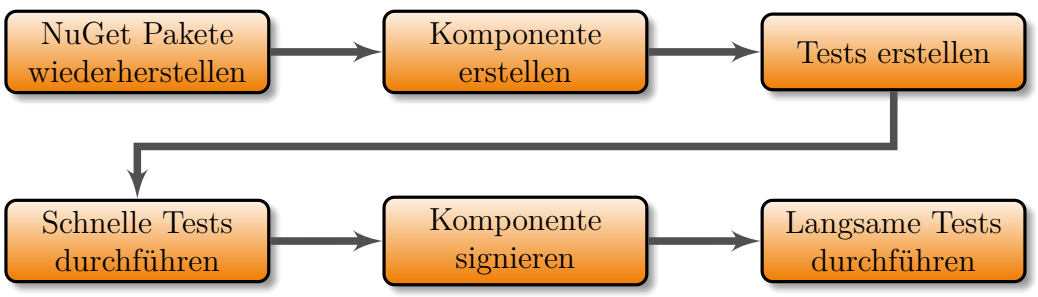

Abbildung 2.6: Ablauf der Komponentenerstellung zur Veröffentlichung

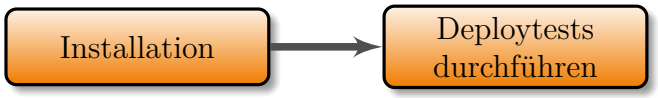

Abbildung 2.7: Ablauf der Deploy Tests

2.7.5 Deploy-Konfiguration

Die in Abbildung 2.7 gezeigte Deploy-Konfiguration wird auf einem System ausgeführt, auf dem die Hostanwendungen installiert sind. Ausgelöst wird diese Konfiguration durch das erfolgreiche Beenden der vorangehenden Build-Konfiguration. Sie führt zunächst die Installation der Komponente durch. Anschließend wird die dritte Testkategorie getestet. Sie umfasst die in Abschnitt 3.4 auf Seite 23 beschriebenen Tests sowie weitere produktspezifische Tests, welche zur Durchführung Zugriff auf die Hostanwendung benötigen.

2.7.6 Deliver-Konfiguration

Die Deliver-Konfiguration hat als einzige Aufgabe das Erstellen und Veröffentlichen der Komponente als NuGet-Paket. Hierzu wird sie durch entsprechende Einträge im SCM-System ausgelöst. Durch ihre spezifizierten Abhängigkeiten zu den beiden vorausgehenden Konfigurationen kann sie erst ausgeführt werden, wenn die restlichen Kettenglieder ausgeführt wurden. Hierdurch ist sichergestellt, dass alle Tests durchgeführt und erfolgreich Beendet wurden.

Kapitel 3

Softwaretests

Die Softwareprodukte der N+P Informationssysteme werden zur Zeit händisch von Kollegen getestet. Dies resultiert in viel Personalaufwand für das Testen und zieht Schwächen in der Testmethodik mit sich. Auch weiterhin sollen Softwaretests durch menschliche Mitarbeiter erfolgen. Diese Tests sollen sich jedoch auf die Bedienbarkeit der Produkte konzentrieren. Rein technische Fehler und Probleme sollen größtenteils durch automatisiert ausgeführte Tests ermittelt werden.

3.1 Ziele

Die Tests der NuPTools verfolgen drei primäre Ziele. Das erste Ziel ist die Qualitätssicherung der Produkte. Menschliche Tester neigen dazu, nicht alle Testfälle zu überprüfen. Dies ergibt sich zum Einen aus der großen Menge an Testfällen, die durch die Anzahl der Eingabewerte entstehen. Zum Anderen übersehen Tester häufig unerwartete Eingaben. Beispielsweise kann ein Feld in einer grafischen Benutzerschnittstelle, das eine Ganzzahl erwartet, auch mit einer leeren Eingabe, einem Text oder einer Gleitkommazahl gespeist werden. Wenn zusätzlich der Wertebereich eingeschränkt ist, ergeben sich bereits fünf Zustände von denen häufig nur ein gültiger getestet wird. Zukünftig sollen solche technischen Aspekte daher automatisiert überprüft werden.

Ein weiteres Ziel ist die automatisierte Testinstallation. Aufgrund der großen Anzahl an Produkten ist eine einzelne Testinstallation von jedem Produkt sowie verschiedener Kombinationen kaum umsetzbar. Durch die Automatisierung sollen Fehler vor der Auslieferung der Software entdeckt werden. Außerdem soll die Fehler auslösende Situation bereits ohne langwierige Analyse eingegrenzt werden.

Das letzte Ziel ist die Verbesserung der Wartbarkeit der Softwarecodes. Hierzu sollen Code-Inspektionen durchgeführt und ausgewählte Metriken berechnet werden. Die Ergebnisse dieser Inspektionen sollen der Schulung des Entwicklerteams dienen und langfristig einen besseren Codestil bewirken.

3.2 Teststufen

Spillner und Linz (2012) beschreibt verschiedene Teststufen anhand des allgemeinen V-Modells. Im Gegensatz zu anderen Modellen wie beispielsweise dem Wasserfallmodell, bei dem das Testen als eine Phase zum Ende des Entwicklungsprozesse gesehen wird, sieht das V-Modell das Testen als gleichwertige Tätigkeit an. Die Tests sind hier zeitlich gesehen immer noch nach der vollständigen Programmierung platziert. Es existiert jedoch zu jedem Detailgrad der Entwicklung eine korrespondierende Testphase. In Abbildung 3.1 sind die Stufen des allgemeinen V-Modells zu sehen.

Den höchsten Detailgrad besitzt die Komponentenspezifikation. Diese Stufe beschreibe das Verhalten einzelner Komponenten. Entsprechend wird die Umsetzung mit Komponententests validiert.

Die nächst höhere Stufe ist der technische Systementwurf. Dieser enthält beispielsweise die Systemarchitektur und Schnittstellendefinitionen. Die korrespondierende Testphase

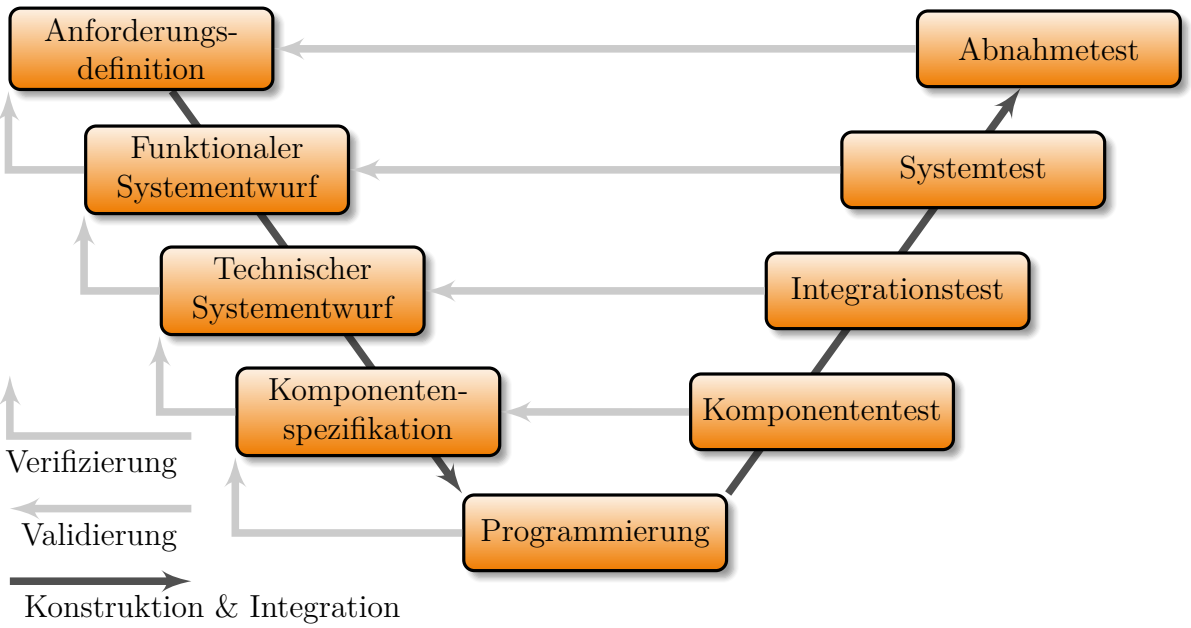

Abbildung 3.1: Allgemeines V-Modell nach Boehm (Spillner und Linz 2012)

hierzu ist der Integrationstest. Diese Stufe testet das Zusammenspiel von einzelnen Komponenten. Der relevante Unterschied zu Komponententests besteht darin, dass Komponententests möglichst ohne die Interaktion mit anderen Komponenten arbeiten sollen. Beim Integrationstest dürfen und sollen die verschiedenen Komponenten interagieren. Dies soll Fehler, die aus Sicht einer einzelnen Komponente fehlerfrei sind, aufdecken.

Die zweithöchste Stufe ist der funktionale Systementwurf. Im Gegensatz zu dem detaillierteren technischen Systementwurf werden hier die funktionalen Komponenten und Abläufe beschrieben. Validiert wird diese Stufe durch den Systemtest. Dieser stellt die allgemeine Funktion und die Korrektheit der Abläufe sicher.

Die höchste Ebene ist die Anforderungsdefinition. Hier sind die Wünsche und Anforderungen des Kunden bzw. Stakeholders definiert. Die Erfüllung der Anforderungen wird durch den Abnahmetest sichergestellt.

3.3 Umsetzung der Tests

Technisch werden die Teststufen im NuPTools-Projekt nur schwach getrennt. Auch werden auf allen Stufen größtenteils die gleichen Testwerkzeuge verwendet. Die unterschiedlichen Tests unterscheiden sich jedoch in der Vorgehensweise und Stilrichtlinien voneinander.

Alle automatisiert durchführbaren Tests verwendet als Basis das Testframework NUnit. Dies ermöglicht die Erkennung und Ausführung der Tests. Je nach zu testendem Detailgrad werden die zu testenden Komponenten jedoch isoliert. Am stärksten findet dies in den Komponententests statt. Hier müssen alle Abhängigkeiten durch kontrollierbare Komponenten ersetzt werden.

Im Rahmen der Überarbeitung des Projektes wurde das Inversion of Control (IoC)-Muster eingeführt. Dieses zeigt in den Tests einen seiner Vorteile. Die Abhängigkeiten werden durch IoC-Komponenten verwaltet und per Constructor Injection eingebunden. In den Tests kann ohne das IoC-Verfahren gearbeitet werden. Dazu werden die benötigten Komponenten als Konstruktorparameter übergeben. Um die gewünschte Isolation der Komponenten zu erreichen, können die zu testenden Komponenten mit Testattrappen der Abhängigkeiten instanziiert werden. Diese haben ein so angepasstes Verhalten, dass keine unerwarteten oder unerwünschten Vorgänge ablaufen. Nur in wenigen Fällen werden eigens zu Testzwecken Komponenten entwickelt. Häufiger werden Komponenten mit dem FakeItEasy-Framework gefälscht um ein in jedem Fall konstantes Verhalten zu erzeugen.

Die zweite Teststufe des V-Modells, die in Form von Integrationstests umgesetzt wird, verwendet je nach Detailgrad die selben Techniken wie die Komponententests. Da bei der Integration die Interaktion der Komponenten im Vordergrund liegt werden jedoch deutlich weniger Komponenten ersetzt. Vorwiegend werden solche Komponenten ausgetauscht, die Kontakt zur Testumgebung herstellen. Beispielsweise Komponenten, die Dateipfade anstelle von Systempfaden auf Pfade innerhalb der Testumgebung umleiten.

Ein grundlegendes Verständnis für die praktische Umsetzung von Komponententests vermittelt Osherove (2013). Osherove geht dabei vorwiegend auf Komponententests ein. Die vorgestellte Praxis kann jedoch für andere Teststufen adaptiert werden. Besondere Beachtung innerhalb des NuPTools-Projektes finden die Richtlinien bzw. Empfehlungen hinsichtlich der Lesbarkeit und Wartbarkeit der Tests. Diese wurden wie in Abbildung 3.2 umgesetzt.

Für jedes Projekt wird ein Testprojekt mit gleichem Namen und dem Anhang „.Tests“ erstellt. Auf eine Trennung der Komponententests und Integrationstests wird vorerst verzichtet, um dem beim Testen unerfahrenen Entwicklungsteam die harte Trennung zwischen den Teststufen zu nehmen.

Innerhalb der Projekte wird primär die Zuordnung nach Klassen angewandt. Somit wird für jede getestete Klasse eine Testklasse mit dem Anhang „Tests“ erstellt. Bei Bedarf kann eine zusätzliche Zuordnung nach Funktionalität erfolgen. Dabei werden, wie in dem Beispiel gezeigt, die Tests einer bestimmten Funktionalität der getesteten Klasse in eine eigene Testklasse ausgelagert.

Für die Testfunktionsnamen wird die in Abbildung 3.3 auf der nächsten Seite gezeigte Syntax als Richtlinie umgesetzt. Diese besteht aus drei Blöcken. Dem Namen der getesteten Funktion, den im Test verwendeten Werten bzw. Zuständen und dem erwarteten

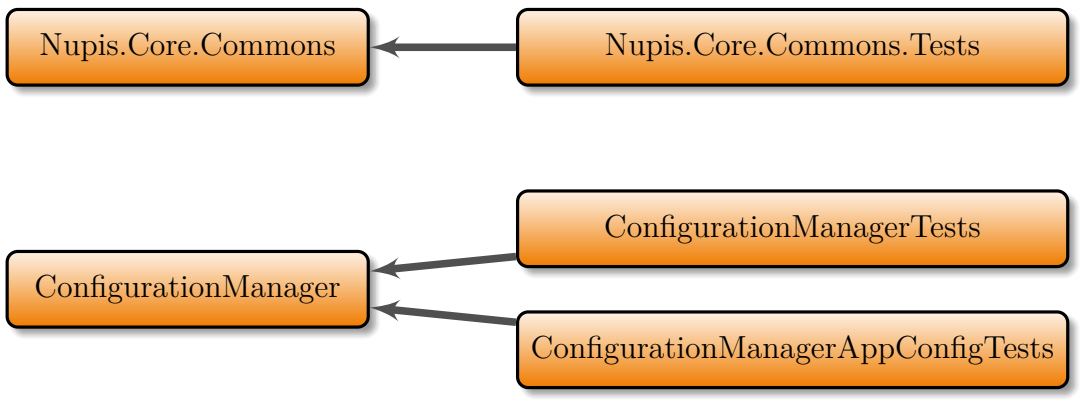

Abbildung 3.2: Testzuordnungen im NuPTools-Projekt

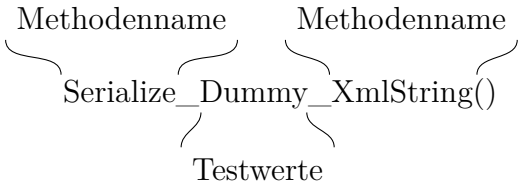

Abbildung 3.3: Syntax der Testfunktionen

Ergebnis.

Diese Struktur erleichtert die Orientierung zwischen den Testergebnissen, Tests und Komponenten. Ein Beispiel für eine Fehlerausgabe in dieser Struktur ist Ausschnitt 3.1. In der Nachricht ist auf einen Blick die Testklasse und damit verbunden die getestete Klasse erkennbar. Außerdem sind die getestete Methode, das erwartete und das tatsächliche Ergebnis sofort zu finden.

```
ConfigurationManagerTests.Serialize_Dummy_XmlString() failed.
   Expected value to be 6, but found 5.
```

Ausschnitt 3.1: Beispielausgabe eines fehlgeschlagenen Tests

3.4 Tests über Grafische Schnittstellen

Zur Umsetzung der Systemtests der dritten Teststufe müssen im NuPTools-Projekt grafische Oberflächen untersucht werden. Hierzu wird das White[1]-Framework verwendet. Das zu testende Plug-in wird durch den Buildserver automatisiert installiert. Die mit NUnit und dem White-Framework ausgeführten Tests können die Hostanwendung inklusive dem Plug-in starten und steuern. Zum Sicherstellen der korrekten Installation und Ausführbarkeit öffnet der Test die Konfigurationsseite des zu testenden Plug-ins. Dies kann nur fehlerfrei ausgeführt werden, wenn alle benötigten Produktkomponenten von der Hostanwendung geladen werden konnten.

Da viele Daten der Hostanwendung sehr komplex sind und das Fälschen der entsprechenden Daten kaum möglich ist, verwenden auch einige Integrationstests die grafische Schnittstelle der Hostanwendung. Auch hier starten die Tests mit Hilfe von White die Anwendung. In den Integrationstests muss die Anwendung jedoch nur wenig gesteuert werden, da viele Funktionen über eine COM-Schnittstelle gesteuert werden können.

[1] http://white.teststack.net/

3.5 Testwerkzeuge

Zur Durchführung der Tests ist ein Test-Framework erforderlich. Hier bietet sich das in Visual Studio integrierte MsTest Framework an. Mauer (2015) und Seemann (2010) nennen einige Schwächen des Frameworks. Zu den Schwächen gehören unter Anderem:

- Langsame Testausführung
- Speicherverbrauch durch fehlerhafte Aufräumprozesse
- Probleme mit verschiedenen Konfigurationsdateien
- Fehlende Erweiterbarkeit bzw. Anpassbarkeit
- Kryptische Testergebnisse

Daher kommt in dem NuPTools-Projekt ein anderes Framework zum Einsatz. Populäre Alternativen sind das NUnit-Framework[2] und das xUnit.net-Framework[3]. Beide Frameworks verfolgen ähnliche Ziele, unterscheiden sich aber in einigen entscheidenden Details. xUnit.net ist als minimalistisches Framework ausgerichtet, während NUnit einen deutlich größeren Funktionsumfang hat. Aufgrund dieses größeren Umfangs werden die Tests der NuPTools mit dem NUnit-Framework umgesetzt.

3.5.1 NUnit

Das NUnit-Framework unterstützt den Testprozess mit einem umfangreichen Werkzeugsatz. Dieser ermöglicht eine flexible Konfiguration von Testfällen und -eigenschaften. Alle Möglichkeiten und Details des Frameworks sind in (NUnit Software 2017) dokumentiert.

3.5.1.1 Kategorisierung mit NUnit

NUnit bietet verschiedene Möglichkeiten zum Kategorisieren von Tests mittels Klassen- und Funktionsattributen. Zum Ausschließen von Tests kann das *Ignore*-Attribut verwendet werden. Dies bewirkt, dass die so markierten Testfunktionen- oder Klassen anstelle der Ausführung nur eine Warnung erzeugen. Zusätzlich kann hierbei ein Datum angegeben werden, bis zu dem diese Markierung gültig ist. So können bereits bestehende Tests erst zu einem bestimmten Datum gültig werden.

[2] https://www.nunit.org/
[3] https://xunit.github.io/

Eine zweite Möglichkeit zum Ausschließen von Tests ist das *Explicit*-Attribut. Dieses bewirkt, dass die hiermit markierten Tests nur ausgeführt werden, wenn Sie explizit über einen Namen oder einen Filter ausgewählt wurden. Ein möglicher Anwendungsfall hierfür sind Tests die einzelne Oberflächenkomponenten anzeigen. Da diese Komponenten auf Benutzereingaben warten und somit nie bzw. nur durch eine Zeitüberschreitung beendet werden ist es sinnvoll sie von der automatisierten Ausführung auszuschließen. Entwickler können die Tests dennoch lokal auswählen und ausführen, um während der Entwicklung einzelne Komponenten anzeigen zu können.

Einen einschließenden Ansatz verfolgen das *Category*-Attribut. Hiermit können einzelnen Tests Kategorien zugeordnet werden. Die Kategoriebezeichnung ist frei wählbar, ein populäres Beispiel sind jedoch die Kategorien „Long Running“ und „Short Running“. Also die Aufteilung der Tests nach kurzen und langen Durchführungsdauern. Bei der Testdurchführung können die auszuführenden Kategorien gewählt werden und so eine oder mehrere bestimmte Arten von Tests ausgewählt werden. Da das Kategorieattribut nicht ausschließend ist, werden ohne spezifizierte Kategorie standardmäßig alle Tests ausgeführt.

3.5.1.2 Konfiguration von NUnit-Tests

Ebenso wie die Kategorisierung erfolgt die Konfiguration der Testeigenschaften mittels Attributen. NUnit bietet hierzu eine Vielzahl von Attributen an. Einige diese Attribute sind von beschreibender Art wie *Author* oder *Description*. Andere Attribute dienen zur Konfiguration der Tests. Beispielsweise kann der *ApartmentState* definiert werden. Standardmaßgig werden Tests im Multi Thread Apartment-Zustand ausgeführt. COM-Objekte, wie grafische Oberflächen benötigen jedoch häufig den Single Thread Apartment-Zustand.

Ebenso lässt sich beispielsweise mit dem *Culture*-Attribut die regionale Kultur des Prozesses definieren oder mit *MaxTime* eine Zeitüberschreitung für die Testausführung setzen.

3.5.1.3 Testfallkonfiguration mit NUnit

Eine wichtige Konfigurationsmöglichkeit ist die Konfiguration der Testfälle. Auch dies wird mit dem NUnit-Framework durch Attribute umgesetzt. Die Möglichkeiten umfassen das Festlegen der Testparameter, Variationen der Kombinationsstrategie der Parameter und Wiederholungsanweisungen.

Attribut	**Beschreibung** **Beispiel**

Values	Mit dem *Values*-Attribut lässt sich eine Auflistung von Testwerten spezifizieren.
	`[Test]`
	`public void Test([Values(1, 2, 3)] int x)`
	`{ ... }`

Range	*Range* erzeugt entsprechend der konfigurierten Grenzen und Schrittweiten numerische Werte. Im folgenden Beispiel werden elf Werte von 1,0 bis 2,0 in 0,1er Schritten erzeugt.
	`[Test]`
	`public void Test([Range(1, 2, 0.1)] int x)`
	`{ ... }`

Random	Mit *Random* können zufällige nummerische Werte entsprechend dem spezifizierten Bereich erzeugt werden. Im folgende Beispiel werden fünf Werte im Bereich von 0,5 und 2,5 erzeugt.
	Für die Generierung der Werte verwendet das NUnit-Framework eigene Komponononenten, die bei jedem Testdurchlauf den selben Startwert verwendet. Dies bewirkt, dass bei jedem Durchlauf die selben Werte erzeugt werden und die Tests trotz der Zufallswerten wiederholbar sind.
	Eingeschränkt wird dieses Verhalten jedoch von der Testanzahl. Da die Reihenfolge der Tests nicht deterministisch ist, die Erzeugung der Werte jedoch von der Anzahl der Verwendungen des Zufallsgenerators abhängt, können die Werte mit veränderter Testanzahl variieren.
	`[Test]`
	`public void Test([Random(0.5, 2.5, 5)] int x)`
	`{ ... }`

Attribut	**Beschreibung** **Beispiel**

TestCase	Mit Hilfe des *TestCase*-Attributes können vollständige Testwertsätze und erwartete Ergebnisse konfiguriert werden. Eine Kombination der Werte aus unterschiedlichen Testfällen findet hierbei nicht statt.
	`[TestCase(12, 3, ExpectedResult=4)]`
	`[TestCase(12, 2, ExpectedResult=6)]`
	`public int Test(int n, int d)`
	`{`
	`  return n / d;`
	`}`

TestCaseSource	*TestCaseSource* erzeugt den gleichen Testablauf wie *TestCase*. Anstelle der Werte wird jedoch eine Methode, ein Feld oder eine Eigenschaft angegeben. Diese erzeugt bzw. beinhaltet die Testfälle. Hierdurch ist auch eine dynamische Erzeugung von Testfällen realisierbar.
	`static object[] Cases =`
	`{`
	`  new object[] { 12, 3, 4 },`
	`  new object[] { 12, 2, 6 }`
	`};`
	
	`[TestCaseSource("Cases")]`
	`public void Test(int n, int d, int q)`
	`{`
	`  Assert.AreEqual(q, n / d);`
	`}`

Attribut	**Beschreibung** **Beispiel**

ValueSource	Ähnlich wie *TestCaseSource* kann mit *ValueSource* eine Methode, ein Feld oder eine Eigenschaft als Quelle angegeben werden. Diese liefert jedoch nur Werte für einen einzelnen Parameter des Tests.
	`public static int[] Values()`
	`{`
	`  return new[] { 1, 2, 3 };`
	`}`
	
	`[Test]`
	`public void Test([ValueSource("Values")] int x)`
	`{ ... }`

Sequential	Durch das *Sequential*-Attribut wird festgelegt, dass die Testwerte in der angegebenen Reihenfolge durchgeführt werden. Eine Kombination der Werte untereinander findet nicht statt. Im folgenden Beispiel werden somit nur drei Tests erzeugt. Wenn für einen Parameter wie im Beispiel weniger Werte spezifiziert sind, wird in den Tests der Standardwert für den entsprechenden Parametertyp verwendet.
	`[Test]`
	`[Sequential]`
	`public void Test([Values(1, 2, 3)] int x,`
	`  [Values("A", "B")] string s)`
	`{ ... }`

Combinatorial	*Combinatorial* definiert, dass jede Kombination der Testwerte ausgeführt werden soll. Das folgende Beispiel erzeugt sechs Testfälle.
	`[Test]`
	`[Combinatorial]`
	`public void Test([Values(1, 2, 3)] int x,`
	`  [Values("A", "B")] string s)`
	`{ ... }`

Attribut	**Beschreibung** **Beispiel**

Pairwise	Das *Pairwise*-Attribut beschreibt eine intelligente Strategie. Hierbei werden mit Hilfe eines heuristischen Verfahrens die Testwerte so kombiniert, dass jedes Wertepaar möglichst nur einmal aber mindestens einmal verwendet wird. Dadurch reduziert diese Strategie die Anzahl der Tests. Im folgenden Beispiel würden mit der Combinatorial-Strategie zwölf Tests generiert werden. Durch das *Pairwise*-Attribut werden jedoch nur sechs Tests benötigt.
	`[Test]`
	`[Pairwise]`
	`public void Test([Values("a", "b", "c")] string a,`
	`[Values("+", "-")] string b,`
	`[Values("x", "y")] string c)`
	`{ ... }`

Repeat	Mit *Repeat* wird die Anzahl der Wiederholungen festgelegt. Die Durchführung der Tests wird jedoch abgebrochen sobald ein Test fehlerhaft endet. Im folgenden Beispiel werden alle Tests drei mal durchgeführt, sofern der jeweilige Test nicht fehlerhaft ist.
	`[Test]`
	`[Repeat(3)]`
	`public void Test([Values(1, 2, 3)] int x)`
	`{ ... }`

Retry	*Retry* dient zur Konfiguration mehrerer Versuche nach dem Fehlerfall. Jeder Test wird so oft wiederholt bis er fehlerfrei ist oder die maximale Anzahl an Durchläufen erreicht ist. Im folgenden Beispiel resultiert dies darin, dass jeder Test mindestens einmal und maximal drei mal durchgeführt wird.
	`[Test]`
	`[Retry(3)]`
	`public void Test([Values(1, 2, 3)] int x)`
	`{ ... }`

3.5.1.4 Testablauf mit NUnit

Eine Testklasse in NUnit wird Fixture genannt. Sie kann beliebig viele Tests enthalten. Abbildung 3.4 zeigt den allgemeinen Ablauf einer Fixture. Die Vorbereitungs(SetUp)- und die Aufräumphase(TearDown) sind jeweils in zwei Abschnitte geteilt. Dies sind ein Abschnitt je Fixture und ein Abschnitt je Test. Es besteht somit eine gemeinsame Vorbereitungsphase und eine Vorbereitungsphase je Test sowie korrespondierende Aufräumphasen.

Es werden somit alle Tests einer Testklasse innerhalb der gleichen Instanz dieser Testklasse ausgeführt. Zum Aufbau der Testsituation können die Vorbereitungsphasen genutzt werden. Dies bietet sich bei Komponententests häufig an, da eine Testklasse hierbei nur ein Testobjekt untersuchen sollte. Daraus ergibt sich, dass jeder Test die gleichen oder zum Teil gleichen Voraussetzungen benötigt. Durch die wiederverwendeten Phasen kann die Testmethode verkleinert werden, was der Übersicht und Verständlichkeit zuträglich ist. Bei Verwendung dieser Phasen sollte jedoch immer berücksichtigt werden, dass diese Methoden von jedem Test verwendet werden. Es muss daher abgewogen werden, welchen Nutzen und welche Laufzeitkosten die hier implementierten Prozesse bringen falls diese nicht von allen Tests benötigt werden.

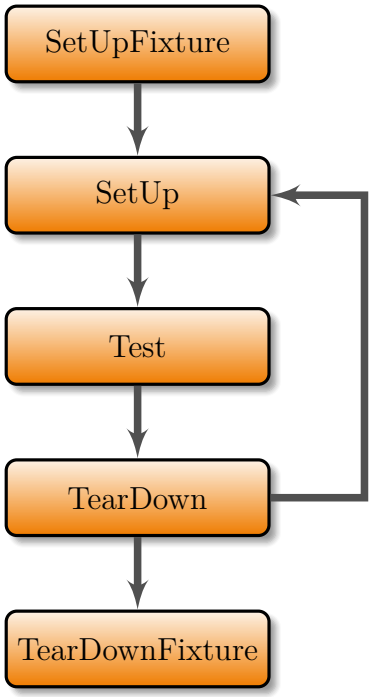

Abbildung 3.4: Allgemeiner Ablauf von NUnit-Testklassen

3.5.2 Fluent Assertions

Das NUnit-Framework wird mit dem Fluent Assertions[4]-Paket kombiniert. Hierbei handelt es sich um eine Sammlung von Erweiterungsmethoden, mit denen die Testannahmen natürlichsprachlicher geschrieben werden können. Das Beispiel in Ausschnitt 3.13 zeigt die gleiche Testannahme mit regulären NUnit-Mitteln und mit Fluent Assertions. Hier zeigt sich, dass die Annahme mit Fluent Assertions der natürlichen Sprache stark ähnelt. Die Tests sind durch die natürlicher wirkende Formulierung leichter zu verstehen und nachzuvollziehen.

```
var testString = "Nupis";

//With regular nUnit assertion
Assert.AreEqual(testString, "Nupis");

//With Fluent Assertions
testString.Should().Be("Nupis");
```

Ausschnitt 3.13: Testannahmen mit NUnit und Fluent Assertions

3.5.3 FakeItEasy

Bei dem FakeItEasy-Framework[5] handelt es sich um ein sog. Mocking-Framework. Mocking-Frameworks dienen zum Erstellen von Testobjekten. Dazu erstellen sie Attrappen der Abhängigkeiten des Testobjektes. Das Verhalten dieser Attrappen kann präzise festgelegt werden, sodass das ursprüngliche Verhalten keinen Einfluss mehr auf das Testobjekt hat.

Ausschnitt 3.14 auf der nächsten Seite zeigt ein einfaches Anwendungsbeispiel für das FakeItEasy-Framework. Es legt zunächt die beiden gefälschten Objekte *lollipop* und *shop* an. Diese implementieren die Schnittstellen *ICandy* und *ICandyShop*. Als nächstes wird für die *GetTopSellingCandy*-Methode des *shop*-Objektes konfiguriert. Durch den hier gezeigten Befehl liefert sie bei jedem Aufruf das *lollipop*-Objekt zurück. Nachdem die gefälschten Objekte konfiguriert wurden, wird das zu testende Objekt erstellt. Der zu testenden Methode wird das gefälschte Objekt als Parameter übergeben. Im Anschluss wird in dem Beispiel mit Hilfe des FakeItEasy-Frameworks das Testkriterium geprüft.

[4] http://fluentassertions.com/
[5] https://fakeiteasy.github.io/

```
var lollipop = A.Fake<ICandy>();
var shop = A.Fake<ICandyShop>();

A.CallTo(() => shop.GetTopSellingCandy()).Returns(lollipop);

var developer = new SweetTooth();
developer.BuyTastiestCandy(shop);

A.CallTo(() => shop.BuyCandy(lollipop)).MustHaveHappened();
```

Ausschnitt 3.14: Anwendungsbeispiel für FakeItEasy (FakeItEasy 2017)

Zum Bestehen dieses Tests muss die *BuyCandy*-Methode des *shop*-Objektes mit dem *lollipop*-Objekt aufgerufen worden sein.

3.5.4 White

White ist ein Framework zum automatisieren von grafischen Anwendungen. Es unterstützt u.A. Win32- und WPF-Anwendungen. Die Tests mit White können mit einer beliebigen .Net-Sprache geschrieben werden. Der grundlegende Ablauf der Oberflächensteuerung ist relativ simple und umfasst die folgenden Schritte:

1. Anwendung starten
2. Anwendungsfenster ermitteln
3. Steuerelement ermitteln
4. Eingabe tätigen
5. Anwendung beenden

Die Schritte 2 bis 4 können je nach Testfall mehrfach auftreten. Beispielsweise beim Öffnen zusätzlicher Fenster oder der Steuerung mehrerer Steuerelemente. Ein beispielhafter Test, in dem diese Schritte umgesetzt werden, ist in Ausschnitt 3.15 auf der nächsten Seite zu sehen.

```
[Test]
[Category("deploy")]
public static void TestConfiguration()
{
    var application = Application.Launch(Path);
    var window = application.GetWindow(AppWindowTitle,
       InitializeOption.NoCache);

    //Switch tab
    var fileButton = window.Get<Button>(
       SearchCriteria.ByAutomationId("nupis.tab"));
    fileButton.Click();

    //Open Config
    var configButton = window.Get<Button>(
       SearchCriteria.ByAutomationId(
       "Nupis.Inventor.NuPTools.Config"));
    configButton.Click();
}
```

Ausschnitt 3.15: Beispielhafter Ablauf eines Tests mit White und NUnit

3.5.5 Nupis.TestFramework

Das firmeneigene TestFramework unterstützt die Softwaretests durch generische Tests. Ein Beispiel hierfür ist das automatische Testen von Konfigurationsobjekten. Diese müssen in XML-Form umwandelbar sein. Alle Konfigurationsobjekte einer zu testenden Komponente werden dazu ermittelt und testweise umgewandelt.

Zusätzlich stellt das TestFramework Hilfsmethoden zur Verfügung, die in den Softwaretests häufig zum Einsatz kommen. Dies Umfasst beispielsweise Methoden zum Öffnen und Verwalten Hostanwendungen.

3.6 Inspektionen

Bei der Inspektion einer Software werden verschiedene Qualitätsmerkmale untersucht. Dies umfasst im wesentlichen Metriken und Codestilanalysen. Im NuPTools Projekt sollen die Inspektionen der Kontrolle und Verbesserung der Qualität dienen. Besonderer Augenmerk liegt dabei auf der Wartbarkeit und damit auch auf der Verständlichkeit des Codes. Für die Inspektion werden die Tools JetBrains ReSharper Command Line Tool (R

CLT)[6], SonarQube[7] und NDepend[8] in den Buildprozess integriert. Der Funktionsumfang der Werkzeuge Überschneidet sich teilweise. In diesen Fällen wird für die jeweiligen Informationen nur eines der Tools primär eingesetzt.

3.6.1 Codestilanalyse

Zum Codestil existieren unzählige Richtlinien und Konventionen. Im .Net-Umfeld sind meist die Richtlinien von Cwalina und Abrams (2008) oder daran angelehnte Konventionen zu finden. Diese von Microsoft zitierten Richtlinien werden häufig als offizielle C#-Richtlinien betrachtet und fanden daher eine entsprechende Verbreitung.

Ebenfalls weit verbreitet sind die von der Visual Studio Erweiterung ReSharper standardmäßig umgesetzten Stilregeln. Auch diese Regeln stimmen größtenteils mit den Richtlinien von Cwalina und Abrams (2008) überein. Aufgrund ihrer Verbreitung und der Ähnlichkeit zu dem von Microsoft empfohlenen Stils können sie als Standard angesehen werden. Da sowohl ReSharper als auch TeamCity vom gleichen Anbieter entwickelt werden ist eine Codestilanalyse nach diesen Regeln bereits im TeamCity-System integriert. Aufgrund der Verbreitung dieser Stilregeln und der vorhandenen Integration des Analysewerkzeugs kommen sie im NuPTools Projekt zum Einsatz.

Die Analyse umfasst eine Vielzahl verschiedener Kriterien. Neben einfachen Verstößen gegen Namenskonventionen werden beispielsweise auch Zugriffsverletzungen aufgezeigt. Beispiele für gefundenen Probleme durch die Inspektion sind:

- Verletzung der Namenskonvention
- *for*-Schleife anstelle einer *foreach*-Schleife
- Nicht erreichbarer Code
- Ungenutze Variable
- Öffentlich zugreifbares Feld

Aufgrund des Umfangs der Analyse ist diese nicht nur der Vereinheitlichung des Stils sondern auch der Effizienz und Fehlertoleranz zuträglich.

[6] https://www.jetbrains.com/resharper/features/command-line.html
[7] https://www.sonarqube.org/
[8] https://www.ndepend.com/

Auch die beiden anderen eingesetzten Werkzeuge können eine solche Stilanalyse durchführen. Primär werden jedoch die Analyseergebnisse des R# CLT verwendet, da die Ergebnisse hier grafisch am übersichtlichsten aufbereitet sind.

3.6.2 Hinweise zu gängigen Praktiken und sprachlichen Verbesserungen

Das R# CLT kann Hinweise zu gängigen Praktiken und programmiersprachlichen Verbesserungen geben. Diese Funktion findet besonders beim Wechsel der Programmiersprachenversion viele Verbesserungsmöglichkeiten und zeigt so die Möglichkeiten der neuen Version. Das NuPTools-Projekt ist über viele Jahre und Programmiersprachenversion herangewachsen. Es sind daher viele solcher veraltete Konstrukte anzutreffen. Durch die Möglichkeiten in neueren Versionen können diese vereinfacht werden. Mithilfe dieser Inspektion lässt sich daher die Lesbarkeit des Codes an vielen Stellen verbessern.

Diese Analyse kann auch mit den anderen Werkzeugen durchgeführt werden. Die untersuchten Regeln variieren jedoch. Das R# CLT stellt hier eine Art Mittelweg dar, der die wichtigsten Regeln umfasst aber auf die abstraktesten Regeln verzichtet. Dies kommt dem Zustand entgegen, dass Teile des Entwicklerteams mit diesen Programmiertechniken nicht vertraut sind.

3.6.3 Fehleranalyse

Im Rahmen der Inspektion wird der Code auch auf Programmfehler untersucht. Hierzu wird die Software SonarQube verwendet. Bei dieser Fehleranalyse handelt es sich um eine statische Fehlersuche. Beispiele für die hierbei gefunden Fehler sind:

- Nicht abgesicherte Methodenparameter, die Fehler auslösen könnten
- Schleifen, die immer im ersten Durchlauf ihre Ausführung beenden
- if-Anweisungen, die immer zu gleichen Ergebnis ausgewertet werden

Einige dieser Fehler können, sofern sie zur Laufzeit auftreten, als kritisch angesehen werden. Zum Teil können hierdurch schwere Konsequenzen in Form von Programmabstürzen oder fatalen Datenfehlern auftreten.

Die bereits häufig veränderten Programmcodes im NuPTools-Projekt weisen einige dieser Fehler auf. Häufig wurden Änderungen ohne vollständiges Verständnisses des Codes und dessen Auswirkungen umgesetzt. So kamen viele Codeabschnitte zustande, die nicht mehr erreicht werden können oder ihr Verhalten signifikant geändert haben.

3.6.4 API Breaking Changes

In einem modularen System wie den Kernkomponenten des NuPTool-Projektes ist es wichtig, dass die Schnittstellen der einzelnen Komponenten abwärtskompatibel bleiben. Beispielsweise kann beim Ändern der Sichtbarkeit einer Methode von öffentlich auf privat aus Sicht der Komponente nicht sichergestellt werden, dass das Fehlen dieser Methode bei verwendenden Anwendungen keine Fehler auslöst. Daher sind solche Änderungen nur mit extremer Vorsicht durchzuführen.

Das Tool NDepend unterstützt Entwickler bei der Vermeidung solcher Fehler mit einer speziellen Analyseart. Hierbei werden Codeelemente mit ihren Vorgängerversionen verglichen. Gefährliche Änderungen an Schnittstellen werden dabei gefunden und zu einem Analysereport zusammengefasst. Die dabei untersuchten Änderungen sind:

- Änderung der Sichtbarkeit von Typen
- Änderung der Sichtbarkeit von Methoden
- Änderung der Sichtbarkeit von Feldern
- Änderungen an öffentlichen *Interfaces* und abstrakten Klassen
- Änderungen an serialisierbaren Typen
- Änderungen an *Flags*

Da in der Vergangenheit kaum Rücksicht auf die Schnittstellen gelegt wurde, kam es häufig zu Problemen bei der Installation der NuPTools. Häufig führte dies dazu, das für einen spezifischen Kunden das gesamte Projekt inklusive der vorhanden Kernkomponenten händisch neu erstellt wurde um die Kompatibilität zwischen den Komponenten sicherzustellen. Um diesen Aufwand in der Zukunft zu vermeiden unterstützt diese Analyse die Entwicklung und Hilft bei der Reduktion solcher Probleme.

3.6.5 Dublettensuche

Bei der Dublettensuche werden mehrfach vorhandene Codeabschnitte gesucht. Diese können die Wartung und Fehlerkorrektur deutlich erschweren. Dem Entwickler ist häufig nicht bewusst, dass die geänderte Codestelle mehrfach existiert und Änderungen somit mehrfach eingepflegt werden müssen.

Für die Dublettensuche wird R# CLT eingesetzt. Neben exakten Kopien erkennt die Software auch einfache Muster. Dies Umfasst beispielsweise Variationen in Klassennamen oder Parameter in Methodenaufrufen. Besonders die Mustererkennung ist anfällig für falsche Fehler. Daher müssen generierte Dateien, AssemblyInfo-Dateien und für die Paketerstellung zusammengestellte Ordner, die auch Quellcodes enthalten, von der Suche ausgeschlossen werden.

Das Dublettenproblem ist im NuPTools-Projekt besonders verbreitet. In der Vergangenheit wurden oft Codeabschnitte kopiert anstatt zu generischen Methoden extrahiert zu werden. Die Analyse bietet eine übersichtliche Auflistung vieler diese Dubletten. Somit kann eine Aufarbeitung der Dubletten erfolgen und die Gefahren, die bisher durch die Dubletten entstand, verringert werden.

3.6.6 Komplexität

Die Komplexitätsmessung dient zur Einschätzung der Verständlichkeit des Codes. Hierfür wird in dem Projekt SonarQube eingesetzt.

Die Software verwendet zur Messung das Verfahren kognitive Komplexität. Campbell (2017) beschreibt das Verfahren im Vergleich zu der zyklomatischen Komplexität. Die zyklomatische Komplexität bewertet die möglichen Durchlaufwege durch eine Methode. Dabei wird für jeden möglichen Weg durch eine Methode der Wert um einen Zähler erhöht. So wird eine technische Komplexität der Methode gemessen. Der Aufwand zum Verständnis der Methode variiert jedoch je nach verwendetem Konstrukt. Nicht jede Verzweigung benötigt zum Verständnis den gleichen mentalen Aufwand. Als Beispiel nennt Campbell den Unterschied zwischen einem *switch*- und einem *if-else*-Konstrukt. Während ein Entwickler bei einem *if-else*-Konstrukt bei jeder Überprüfung mentalen Aufwand zum Auswerten des Ausdrucks erbringen muss, erbringt er diesen Aufwand bei einer *switch*-Anweisung nur für den anfänglichen Ausdruck.

Um mit der Bewertung dem mentalen Aufwand zu entsprechen folgt die kognitive Komplexität den folgenden Prinzipien:

- Verkettete Ausdrücke ignorieren
- Werten von Brüchen im Kontrollfluss
- Werten von Brüchen durch unterschiedliche logische Operatoren in verketteten Ausdrücken
- Werten von Rekursionen
- Werten von Sprung- und Unterbrechungsmarken
- Verschachtelungstiefe nimmt Einfluss auf die Wertung

Mit verketteten Ausdrücken sind Konstrukte von modernen Programmiersprachen gemeint, die eine Kurzschreibweise für längere Konstrukte ermöglichen. Die Ausschnitte 3.16 und 3.17 zeigen Beispiele eines *null*-bedingten Zugriff als klassische Variante und als Kurzschreibweise. Hier ist zu sehen, dass die Kurzschreibweise deutlich leichter zu verstehen ist.

Als ein Bruch im Kontrollfluss sieht Campbell alles, was den Kontrollfluss von einer geraden Linie ablenkt. Dies umfasst beispielsweise Bedingungsanweisungen, Schleifen und Fehlerbehandlungen. Auch die zyklomatische Komplexität wertet diese Brüche aus. Der Unterschied besteht jedoch darin, dass die zyklomatische Komplexität jeden möglichen Fall wertet, während die kognitive Komplexität nur die Entscheidungen bewertet. Dieser Unterschied macht sich beispielsweise in dem anfangs genannten *if-else*/*while*-Beispiel bemerkbar, aber auch in einer einfachen *if-else*-Anweisung. Nach dem zyklomantischen Verfahren würde diese Anweisung mit den zwei Fällen bereits zweifach gewertet werden, im kognitiven Verfahren jedoch nur einfach.

Das nächste Prinzip definiert das Wertungsverhalten von logischen Ausdrücken. Bei der Verkettung von logischen Ausdrücken steigt der Interpretationsaufwand mit gleichen Ausdrücken nur unwesentlich. Deutlich höher ist der Aufwand wenn, wie in der zweiten Zeile des Ausschnitt 3.18 auf der nächsten Seite zu sehen, verschiedene Ausdrücke kombiniert

```
MyObj myObj = null;
if(a != null) {
  myObj = a.myObj;
}
```

Ausschnitt 3.16: Beispiel eines klassischen *null*-bedingten Zugriffs (Campbell 2017, S. 5)

```
MyObj myObj = a?.myObj;
```

Ausschnitt 3.17: Beispiel eines *null*-bedingten Zugriffs in Kurzschreibweise (Campbell 2017, S. 5)

```
a && b && c && d
a && b || c && d
```

Ausschnitt 3.18: Beispielhafte Ketten von logischen Operatoren (Campbell 2017, S. 6)

werden. Die kognitive Komplexität wertet daher die erste Zeile des Ausschnitts nur einmalig, die zweite Zeile wegen der zwei Wechsel (&& zu || zu &&) jedoch dreifach.

Bei der kognitiven Komplexität werden auch Rekursionen berücksichtigt. Diese bei der zyklomatischen Komplexität vollständig ignorierten Konstrukte bedürfen oft besonderer Aufmerksamkeit. Besonders indirekte Rekursion, die über mehrere Methoden verteilt und damit besonders unauffällig sind, steigern den mentalen Aufwand zum Verständnis der jeweiligen Methode.

Sprung- und Unterbrechungsmarken wie beispielsweise Programmsprünge oder Schleifenabbrüche werden im Verfahren von Campbell als Brüche im Kontrollfluss angesehen. Somit werden sie ebenfalls gewertet. Methodenausstiege werden jedoch als Vereinfachung der Methode angesehen. Durch vorzeitige Methodenausstiege wird Code häufig lesbarer und Verschachtlungen können vermieden werden. Sie werden daher positiv angesehen und nicht gewertet.

Einen großen Unterschied zu der zyklomatischen Komplexität macht das letzte Prinzip, die Verschachtelungstiefe. Durch häufiges Verschachteln wird Code schnell unleserlich. Dies wirkt sich negativ auf den mentalen Aufwand aus. Das Verfahren berücksichtigt daher bei jeder Wertung die Verschachtelungstiefe und addiert diese zusätzlich zum Komplexitätsmaß hinzu. Das Beispiel in Ausschnitt 3.19 zeigt welchen Einfluss die Tiefe auf das Maß hat.

```
void myMethod () {
    try {
        if (condition1) {                    // +1
            for (int i = 0; i < 10; i++) {   // +2 (nesting=1)
                while (condition2) { ... }   // +3 (nesting=2)
            }
        }
    } catch (ExcepType1 | ExcepType2 e) {    // +1
        if (condition2) { ... }              // +2 (nesting=1)
    }
}                                       // Cognitive Complexity 9
```

Ausschnitt 3.19: Beispiel für die Verschachtelungstiefe (Campbell 2017, S. 8)

```
int sumOfPrimes(int max) {
    int total = 0;
    OUT: for (int i = 1; i <= max; ++i) {    // +1
        for (int j = 2; j < i; ++j) {        // +2
            if (i % j == 0) {                // +3
                continue OUT;                // +1
            }
        }
    total += i;
    }
    return total;
}                          // Cognitive Complexity 7
                           // Cyclomatic Complexity 4
```

Ausschnitt 3.20: Eine Komplexe Beispielmethode (Campbell 2017, S. 9)

Um Besonderheiten einzelner Programmiersprachen gerecht zu werden, definiert die kognitive Komplexität Ausnahmen. Diese Umfassen zur Zeit jedoch nur jeweils eine Ausnahme für die Sprachen COBOL und JavaScript. Somit sind die Ausnahmen für das NuPTools-Projekt nicht relevant.

Das Verfahren kognitive Komplexität ähnelt dem Verfahren zyklomatische Komplexität. Durch kleine Änderungen am Wertungsverhalten ändert sich jedoch die gemessene Information. Wie in den Ausschnitten 3.20 und 3.21 auf der nächsten Seite zu sehen ist das neuere Verfahren in der Lage die Komplexität von Methoden deutlich ähnlicher zum menschlichen Empfinden zu bewerten. Wenn die Messung, wie im NuPTools-Projekt, die Wartbarkeit verbessern soll, ist dieses Verfahren als Indikator für die Verständlichkeit des Codes besser geeignet als das verbreitete Verfahren zyklomatische Komplexität.

3.6.7 Technische Schulden

Im Rahmen der Inspektion werden auch technische Schulden berechnet. Der Begriff wurde ursprünglich als Metapher von Ward Cunningham eingeführt. Cunningham (2011) erklärt den Begriff. Seine Metapher sieht Entwicklungszeit als Zahlungsmittel. Um ein Softwareprodukt fertigzustellen muss Zeit in dieses investiert werden. Verschiedene Wege in der Entwicklung können Zeit sparen und so für eine frühere Fertigstellung sorgen. Die ersparte Zeit ist jedoch nur geliehen. Um diese Schulden zurückzuzahlen muss der ersparte Aufwand nachgeholt werden.

Ein scheinbarer Weg diese Schulden nicht zahlen zu müssen ist, die behelfsmäßigen Konstrukte weiter zu verwenden. Diese sind jedoch nicht so flexibel, wie es ein langfristiges

```
String getWords(int number) {
    switch (number) {                          // +1
        case 1:
        return "one";
    case 2:
        return "a couple";
    case 3:
        return "a few";
    default:
        return "lots";
    }
 }                        // Cognitive Complexity 1
                          // Cyclomatic Complexity 4
```

Ausschnitt 3.21: Eine einfache Beispielmethode (Campbell 2017, S. 9)

Projekt benötigt um wartbar zu bleiben. Durch diese Inflexibilität wird Aufwand an anderen Stellen erzeugt. Dieser Aufwand kann ähnlich wie Zinsen gesehen werden. Durch das Leihen der Entwicklungszeit ist das Projekt also gezwungen zusätzliche Zeit zu investieren.

Wie auch in der Finanzwelt zerren diese Zinsen an der Zahlkraft des Projektes oder Unternehmens. Je mehr technische Schulden aufgebaut werden, umso mehr Zeit benötigen die Entwickler zum Handhaben der behelfsmäßigen Konstrukte. Im Extremfall kann dies das Projekt, entsprechend der Metapher, zahlungsunfähig gegenüber Änderungen machen. Die Entwicklungsgeschwindigkeit sinkt somit stark.

Mit SonarQube können technische Schulden ermittelt werden. Hierzu ermittelt das Tool die bereits in den Abschnitten 3.6.1 auf Seite 34, 3.6.2 auf Seite 35 und 3.6.3 auf Seite 35 beschriebenen Probleme. Für jedes dieser Probleme ist im System ein Aufwandswert hinterlegt. Diese werden addiert und ergeben die ermittelten technischen Schulden. Aufgrund dieses Verfahrens ergeben sich einige Messungenauigkeiten. Die Inspektionen wird zum Teil primär von R# CLT durchgeführt und die Entwickler richten sich nach diesen Ergebnissen. Die gefundenen Probleme der beiden Tools können jedoch Unterschiede aufweisen. Somit werden die technischen Schulden nicht aus den Analyseergebnissen berechnet, die von den Entwicklern verwendet werden. Da die Tools jedoch ähnliche Ergebnisse liefern wird diese Messungenauigkeit als marginal betrachtet. Einen größeren Einfluss haben die vordefinierten Aufwandswerte. Diese wurden vom Hersteller vordefiniert. Der tatsächlichen Einfluss der Probleme ist von Faktoren wie der Erfahrung des bearbeitenden Entwicklers und der vorhandenen Kenntnis über den Code abhängig. Somit ist der Aufwand praktisch nicht exakt abschätzbar. Die hierdurch entstehenden Abweichungen können sich in unbekannter Stärke sowohl positiv als auch negativ auswirken. Weiterhin bezieht das

Verfahren die architekturelle Qualität, beispielsweise in Form der Komplexität, nicht in die Messung mit ein.

Das NuPTools-Projekt leidet stark unter technischen Schulden. Kleine Änderungen erzeugen aufgrund der bisher nicht durchgeführten Abstraktion häufig das vielfache ihres eigentlichen Aufwandes. Die Belastung durch die technischen Schulden bindet bereits viel Entwicklungszeit für die Erhaltung des aktuellen Standes und Anpassungen an äußere Einflüsse.

Aufgrund dieses Problems sind die technischen Schulden eine wertvolle Metrik für das Entwicklungsteam. Der Wert dient zur Messung eines der größten Probleme des Projektes. Da der ermittelte Wert keine präzise Messung ist wird er als Richtwert angesehen. Als solcher wird er trotz der Ungenauigkeit im NuPTools-Projekt gemessen. Durch seine Darstellung, in Form von Arbeitsstunden und -tagen, ist er leicht verständlich und kann im Management zur Darstellung der aktuellen Lage verwendet werden. Auch für Entwickler wirkt dieser Wert fassbar. Daher wird die Reduzierung des Wertes als erfassbares Ziel verwendet.

3.6.8 Sonstige Metriken

SonarQube und NDepend ermitteln eine große Menge an verschiedenen Metriken. Viele davon dienen der weiteren Berechnung und werden daher benötigt. Für das NuPTools-Projekt haben diese Messwerte jedoch keinen Nutzen bzw. werden nicht verwendet, um die Entwickler nicht mit der Informationsmenge zu überfluten. In Einzelfällen können diese Werte dennoch von Interesse sein. Unter anderem stehen die folgenden Metriken zur Verfügung:

- Codezeilen
- Methodenanzahl
- Klassenanzahl
- Dateianzahl
- Kommentarzeilen
- Anteil an Kommentarzeilen
- Abstraktheitsgrad

- Intermediate Language (IL)-Codezeilen
- Abhängige Typen

3.6.9 Anwendung der Metriken

Das so zusammengestellt System aus R# CLT, SonarQube und NDepend wertet sieben Informationen über das NuPTools-Projekt aus. Präsentiert werden diese Informationen zum Teil direkt im TeamCity-Build-Bericht und zum Teil auf einer eigenständigen Seite von SonarQube. Im TeamCity-Bericht werden die Stilanalyse, gängige Praktiken und Dubletten aufgezeigt. Auch die API-Analyse von NDepend wird in diesen Bericht eingefügt. Die von SonarQube ermittelten Ergebnisse für die Fehleranalyse, Komplexität und technische Schulden werden auf der systemeigenen Seite von SonarQube präsentiert. Hier werden die Informationen auch in neue und bestehende Probleme unterteilt.

3.6.10 Aufarbeitung der Qualitätsprobleme

Das NuPTools-Projekt weißt eine große Menge an Problemen auf. Diese wirken sich unter Anderem auf die technischen Schulden aus. Der Zustand des Projektes soll daher verbessert werden. Diese Aufarbeitung wird als Teamarbeit mit zwei Entwicklern durchgeführt.

Zusätzlich sollen die restlichen Teammitglieder zeitweise an der Aufarbeitung teilnehmen. Diese Gruppenarbeit soll ein besseres Verständnis für die Probleme herzustellen und Programmiertechniken für wartbaren Code vermitteln. Außerdem sollen die Teammitglieder hierbei den Umgang mit den Systemen erlernen.

Kapitel 4

Installations- und Aktualisierungsverfahren

Die NuPTools wurden bisher von Dienstleistern händisch auf Kundensystemen installiert. Dies erzeugt aufgrund diverser Eigenarten bei der Installation regelmäßig schwer nachvollziehbare Probleme. Für den Kunden erzeugt die händische Installation zusätzlich zu den Produktlizenzen Aufwand und Kosten. Häufig werden die Systeme daher gar nicht oder nur bei akuten Problem aktualisiert.

Die NuPTools erhalten daher ein automatisches Installations- und Aktualisierungsverfahren. Durch das Verfahren sollen der Pflegeaufwand und die Installationsprobleme verringern werden.

4.1 Anforderungen

Die verschiedenen Eigenschaften des NuPTools-Projektes erzeugen eine Reihe außergewöhnlicher Anforderungen.

- Das Verfahren soll sowohl Online- als auch Offline ausführbar sein. Hierbei soll die Möglichkeit gegeben werden, die Installationsdaten im Kundennetzwerk zu speichern. Dies soll die Dateien Rechnern ohne Internetanschluss zur Verfügung stellen. Außerdem soll hierdurch die Auslastung des Updateservers verringert werden, da mehrere Installationen innerhalb eines Netzwerkes die Dateien nur einmalig laden müssen.

- Die Einbindung in Installationsskripte muss ermöglicht werden. Hierzu muss das Installationsverfahren über die Kommandozeile steuerbar sein.
- Das Verfahren muss die speziellen Anforderungen in die Versionierung berücksichtigen. Da die verschiedenen Versionen der Hostanwendungen unterschiedliche Schnittstellen bereitstellen, muss sich das Updateverfahren an diesen Versionen orientieren und nur kompatible Produktversionen installieren.
- Häufig verwenden die Anwender mehrere NuPTools. In diesem Fall soll die Aktualisierung alle Produkte gemeinsam Aktualisieren um den Anwender nicht durch unzählige aufeinanderfolgende Aktualisierungen zu blockieren. Außerdem müssen Versionskonflikte vermieden werden. Dazu müssen gemeinsam genutzte Komponenten aller installierten NuPTools den selben Stand haben.
- Die Produktlizenzen der NuPTools sind eine Kombination aus Kauf- und Mietlizenzen. Dies bedeutet, dass die Anwender während ihrer Mietlizenz alle Aktualisierungen erhalten. Nach Ende der Mietdauer besitzen sie das Recht, die letzte Version in ihrem Mietzeitraum weiterhin zu verwenden. Somit muss die Installations- und Aktualisierungsroutine den Lizenzstatus überprüfen und entsprechen diesem Status die Version auswählen.
- Um den Widererkennungsfaktor der NuPTools zu verbessern, soll die Installations- bzw. Aktualisierungsoberfläche an die Corporate Identity der Firma angepasst werden. Ebenso soll die Oberfläche hochwertig wirken und dem Benutzer einen professionellen Eindruck vermitteln.

4.2 Aktualisierungsverfahren

Sowohl zur Installation als auch zur Aktualisierung von Software existieren verschiedene Systeme. Die Systeme können in drei Kategorien unterschieden werden. Installationssysteme wie das WiX Toolset[1] ermöglichen die Erstinstallation einer Software. Häufig unterstützen diese System jedoch keine Aktualisierung bzw. nur in der Form einer erneuten Installation. Sie bieten daher auch keine Funktionen zum Laden der Aktualisierungen oder Benachrichtigen über neue Versionen an.

Andere Systeme bieten reine Aktualisierungsverfahren. Ein Beispiel hierfür ist das NAppUpdate-Framework[2]. Mit diesen Systemen sind keine Erstinstallationen möglich.

[1] http://wixtoolset.org/
[2] https://github.com/synhershko/NAppUpdate

Häufig bieten diese System die Aktualisierung im Hintergrund, also während der Verwendung der Software an. Die dennoch nötigen Benutzerinteraktionen geschehen jedoch häufig über einfach gehaltene Oberflächen.

Die dritte Kategorie sind Paketmanager wie Chocolatey[3] oder NuGet. Ihre Stärke kommt besonders bei modularen Anwendungen, die wie die NuPTools mehrere Einzelkomponenten verwenden, zum Vorschein. Paketmanager verarbeiten vom grundlegenden Prinzip mehrere verschiedene Einzelkomponenten nacheinander. Die Installation mehrere Produkte ist für diese Systeme daher keine Besonderheit.

Viele dieser Systeme ermöglichen verschiedene Anforderungen. Die Kombination der speziellen Anforderungen des Projektes bereiten jedoch Schwierigkeiten. Besonders die Anforderungen an die Versionsierung sind eine Hürde für viele Systeme. Auch die Aktualisierung mehrerer Produkte in einem Vorgang wird von den meisten Systemen nicht unterstützt. Chocolatey kann in einer kostenpflichtigen Lizenzversion alle Anforderungen erfüllen. Das Lizenzmodel des Systems berücksichtigt die Verwendung als integriertes Aktualisierungsverfahren jedoch nicht. Es müsste daher für jeden Anwender eine Lizenz erworben werden. Für das NuPTools-Projekt sind diese Kosten zu hoch.

Für das NuPTools-Projekt wird daher ein eigenes Aktualisierungsverfahren auf Basis von NuGet entwickelt. Mit NuGet lassen sich alle Anforderungen erfüllen. Zusätzlich ist das NuGet-System für Entwicklungszwecke bereits in den Entwicklungsprozess integriert. Die Erweiterung zur Produktauslieferung stellt daher keinen nennenswerten Aufwand dar.

4.3 Umsetzung eines eigenen Aktualisierungsverfahren

Das Aktualisierungsverfahren wird prototypisch auf Basis von NuGet umgesetzt. Für das Beziehen von NuGet-Paketen steht die von Microsoft veröffentlichte NuGet Api v3[4] zur Verfügung. Mit dieser werden zwei Teilkomponenten des Verfahrens implementiert.

Die erste Teilkomponente ist ein Hintergrundlader, der während der Nutzung eines Plugins bzw. der Hostanwendung die installierte Version mit dem Aktualisierungsserver abgleicht. Bei verfügbaren Aktualisierungen werden die neuen NuGet-Pakete zunächst in einem Zwischenspeicher gesucht. Dieser kann lokal auf den Anwenderrechner oder an einem Netzwerkort liegen. Sollte ein Paket nicht vorhanden sein, wird dieses im Hintergrund

[3] https://chocolatey.org/
[4] https://docs.microsoft.com/en-us/nuget/api/nuget-api-v3

heruntergeladen. Anschließend wird der Benutzer über die verfügbare Aktualisierung informiert.

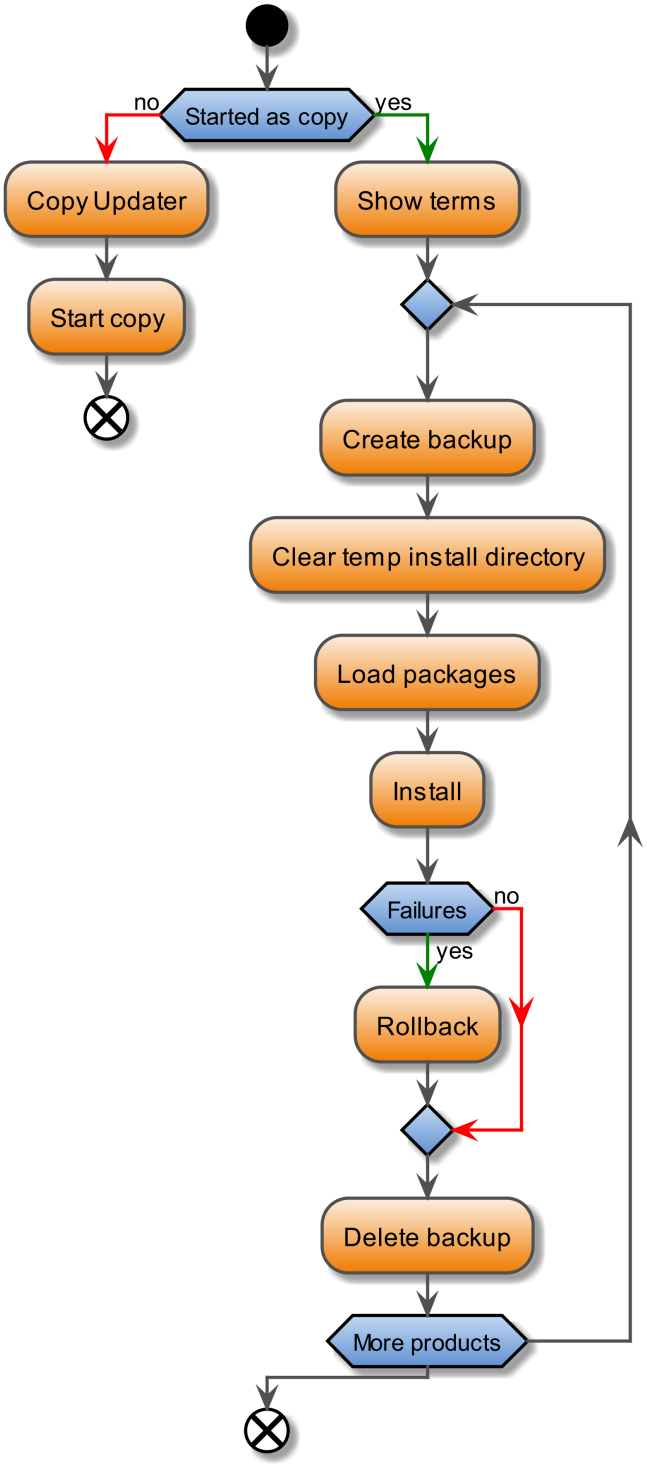

Abbildung 4.1: Ablauf des Aktualisierungsprozesses

Die zweite Teilkomponente ist eine Installationsanwendung für die geladenen Pakete. Sie kann als Aktualisierungsverfahren eigenständig oder unterstützt durch den Hintergrundlader verwendet werden. Ebenso kann sie als Installationsroutine verwendet werden. Sie folgt dem in Abbildung 4.1 auf Seite 47 gezeigten Ablauf.

Um die Aktualisierung der Aktualisierungsanwendung selbst zu ermöglichen, kopiert die Anwendung sich zunächst selbst. Zur Durchführung der Installation wird diese Kopie verwendet, sodass die ursprünglichen Dateien nicht durch die Nutzung schreibgeschützt sind.

Nachdem der Nutzer über die allgemeinen Geschäftsbedingungen informiert wurde, beginnt die eigentliche Installation. Diese wird je Produkt durchgeführt. Der Prozess beginnt mit einer Sicherung der aktuellen Version. Anschließend wird ein temporäres Installationsverzeichnis erstellt bzw. geleert. In dieses werden die benötigten Pakete installiert. Die Pakete werden dabei, wie beim Hintergrundlader auch, zuerst im Zwischenspeicher gesucht. Fehlende Pakete werden von dem Aktualisierungsserver nachgeladen. Bei der Paketinstallation entsteht die in Abbildung 4.2 beispielhaft gezeigt Dateistruktur. Für jedes NuGet-Paket wird eine eigene Ordnerstruktur erstellt. Diese enthält unter Anderem das Paket selbst.

Im letzten Schritt wird aus jedem Paketordner die Produktdateien in den Produktinstal-

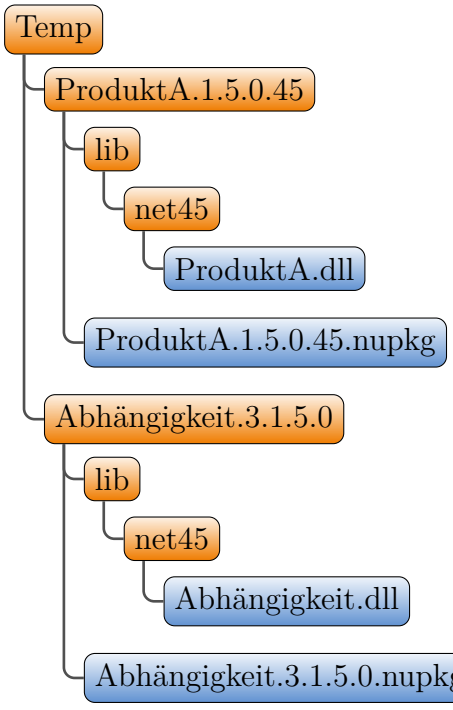

Abbildung 4.2: Beispielhafte Ordnerstruktur eines installierten NuGet-Paketes

lationsordner kopiert. In dem gezeigten Beispiel sind dies die Dateien „ProduktA.dll“ und „Abhängigkeit.dll“. Sofern während der Installation des Einzelproduktes keine Probleme entstanden sind, wird zum Abschluss die gesicherte Produktversion gelöscht.

Kapitel 5

Architektur

Die bestehende Architektur des NuPTools-Projektes ist frei nach Bedarf gewachsen. Sie ist daher kaum strukturiert. Auch eine Dokumentation der Architektur wurde nie erstellt. Im Rahmen der Modernisierung wird die Architektur dokumentiert und analysiert. Ausgehend von dieser Analyse wird die Architektur überarbeitet. Um eine Übersicht über die Architektur zu behalten wird für Teile der Dokumentation ein Generator entwickelt.

5.1 Grundlagen

Die grundlegenden Architekturdokumentationstechniken werden von Starke (2015) erklärt. Vertiefende Informationen zu den verwendeten UML-Diagrammarten erläutert Rupp und Queins (2012). Die Architektur von Softwaresystemen wird zum besseren Verständnis in unterschiedlichen Sichten dokumentiert. Üblicherweise sind diese Sichten Kontextabgrenzung, Laufzeitsichten, Bausteinsichten und Verteilungssichten. Diese Sichtarten sind nicht bindend und können von speziellen Sichten ergänzt werden. Da die Sichten bei Änderungen am System ebenfalls angepasst werden müssen, zieht die Pflege der Sichten entsprechenden Aufwand nach sich. Der Umfang der erstellten Sichten sollte sich daher nach dem Bedarf des Projektes bzw. Auftraggebers richten. Nicht erforderliche Sichten und Sichten mit sehr geringem Informationsgehalt sollten daher vermieden werden.

5.1.1 Kontextabgrenzung

Die Kontextabgrenzung beschreibt die Umgebung des Systems und dessen Interaktion mit den umgebenden Komponenten und Systemen. Ein Beispiel für eine Kontextabgrenzung

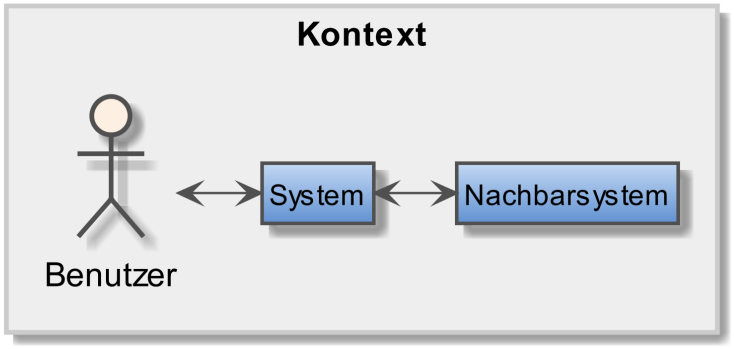

Abbildung 5.1: Beispiel einer Kontextabgrenzung

zeigt Abbildung 5.1. Da es sich bei den NuPTools in der Regel um einfache Funktionserweiterungen in der Hostanwendung handelt, sind Interaktionen zwischen den Plug-ins und der Umgebung, abgesehen von der Hostanwendung selbst, sehr gering. Diese Sicht wird daher für das NuPTools-Projekt nicht erstellt.

5.1.2 Verteilungssicht

Die Verteilungssicht beschreibt die Verteilung des Systems auf Hardwarekomponenten. Das Beispiel in Abbildung 5.2 auf der nächsten Seite zeigt eine exemplarische Entwicklungsumgebung. Hier sind verschiedene virtuelle Systeme ineinander verschachtelt. Zusätzlich wird ein verbundener Entwicklungsrechner gezeigt.

Die NuPTools werden bis auf einzelne Ausnahmen, wie den Lizenzierungsservice, nur lokal auf dem Anwenderrechner ausgeführt. Eine Verteilungssicht beinhaltet daher nur wenige Informationen und wird für das NuPTools-Projekt ebenfalls nicht erstellt.

5.1.3 Bausteinsicht

Eine Bausteinsicht beschreibt das Zusammenspiel von den Systembausteinen. Dies können Komponenten, Pakete oder einzelne Klassen sein. Diese Komponenten können als Whitebox, also mit Sicht auf ihre inneren Komponenten, oder als Blackbox, ohne Sicht auf innere Komponenten dargestellt werden. In Abbildung 5.3 auf der nächsten Seite ist die Komponente *Product* beispielsweise als Whitebox dargestellt. Die inneren Komponenten, wie die *Cache*-Komponente sind als Blackbox dargestellt. Das Beispiel abstrahiert die Komponente auf ihren Namen bzw. Funktion und gibt keine Informationen über den

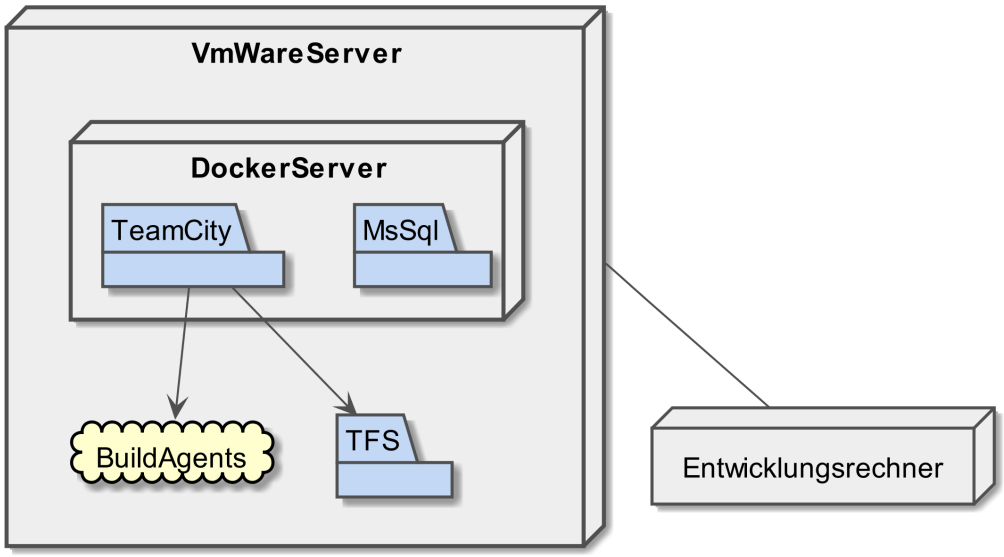

Abbildung 5.2: Beispiel einer Verteilungssicht

inneren Aufbau. Auf diese Weise können Sichten vereinfacht werden. Für den inneren Aufbau der als Blackbox dargestellten Komponenten können bei Bedarf separate Sichten erstellt werden. Da der Umfang an potentiellen Elementen in einer Bausteinsicht sehr groß werden kann, ist dieses Vorgehen empfehlenswert.

Starke (2015) empfiehlt zur Erstellung einer Bausteinsicht ein UML-Klassendiagramm zu verwenden. Bei Bedarf kann dieses mit Komponenten- und Paketsymbolen erweitert werden. Die Komponentensymbole werden dabei für funktionale Komponenten verwendet.

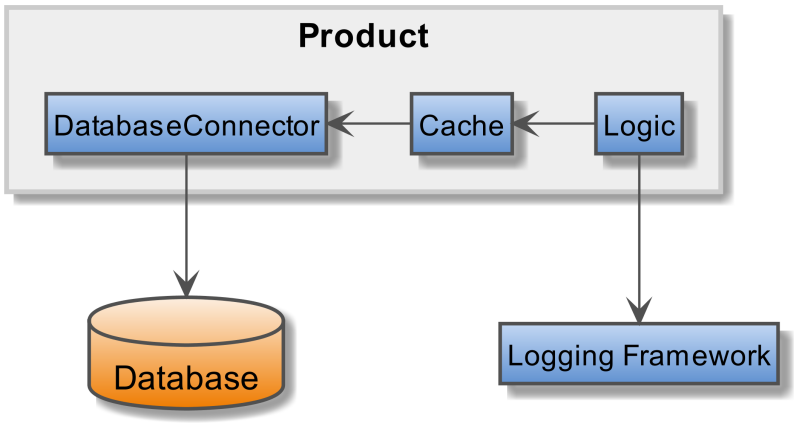

Abbildung 5.3: Beispiel einer Bausteinsicht

Zur Laufzeit sollten Instanzen der so dargestellten Komponenten entstehen. Bei Bedarf kann diese Sicht zusätzlich Details über die angebotenen Schnittstellen darstellen.

Für weniger physische Komponenten, die keine Instanz zur Laufzeit erhalten, können Pakete verwendet werden. Diese repräsentieren eine logische Gruppierung. Beispiele hierfür sind lose Sammlungen an Hilfsfunktionen oder Mechanismen, bei denen viele Einzelkomponenten interagieren und eher der funktionale Effekt anstelle einer greifbaren Komponente von Interesse ist.

5.1.4 Laufzeitsicht

Eine Laufzeitsicht zeigt einen Teilaspekt des entwickelten Systems zur Laufzeit. Sie dient dazu den Ablauf oder die Interaktionen zwischen einzelnen Komponenten darzustellen. Zur Erstellung stehen verschiedene UML-Diagrammarten, wie Sequenz-, Ablauf- oder Kollaborationdiagramme, zur Verfügung. Je nach dargestelltem Aspekt kann zwischen den verschiedenen Arten gewählt werden.

Laufzeitsichten stellen nicht die Struktur, sonder das Verhalten des Systems dar. Sie sind daher besonders anfällig für Änderungen bzw. Anpassungsbedarf nach Änderungen an der Software. Im NuPTools-Projekt werden daher nur wenige Laufzeitsichten erstellt. Zu den erstellten Sichten gehören als besonders wichtig oder kompliziert eingeschätzte Abläufe. Ein Beispiel hierfür ist der Ablauf der Lizenzaktivierung.

5.1.5 Paketdiagramm

Das Paketdiagramm ist einer Bausteinsicht sehr ähnlich. Es konzentriert sich jedoch vorwiegend auf die Codestruktur eines Projektes. Das Beispiel aus Abbildung 5.4 auf der nächsten Seite zeigt die Pakete bzw. Namespaces. Die Pakete werden entsprechend ihrer Anordnung ineinander verschachtelt.

Im Laufe der Zeit sind im NuPTools-Projekt viele Hilfsmethoden entstanden. Diese Methoden wurden nach Bedarf von verschiedenen Entwicklern hinzugefügt. Aufgrund fehlender Richtlinien zur Paketstruktur ist eine gewisse Unordnung entstanden. Als Leitfaden für die Struktur wird ein Paketdiagramm erstellt. Dieses soll die Struktur definieren und darstellen. Mit Hilfe dieses Leitfadens können die Entwickler entsprechende Hilfsmethoden finden bzw. einordnen.

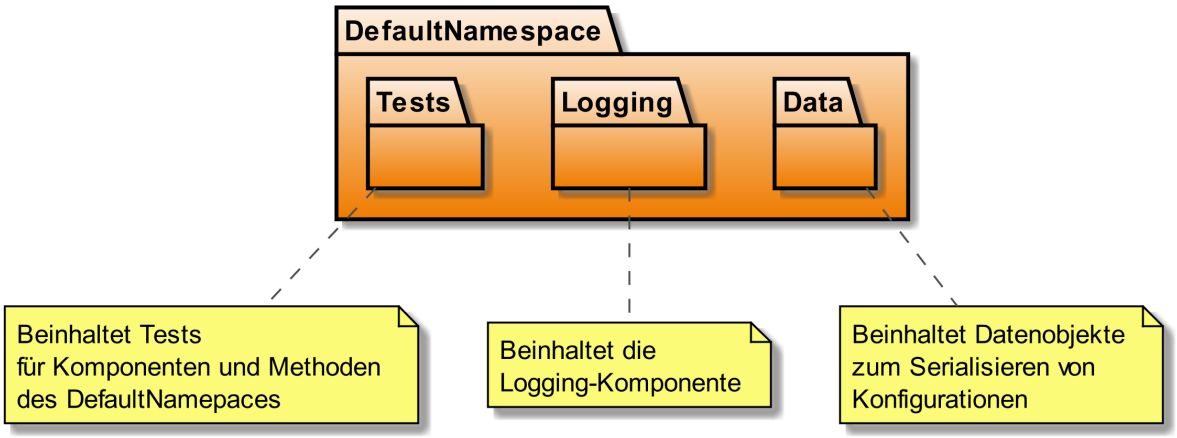

Abbildung 5.4: Beispiel eines Paketdiagramms

5.2 Modellierungsmethoden

Zur Architekturdokumentation bzw. -modellierung existieren verschiedene Werkzeugarten. Im Wesentlichen lassen sich die Möglichkeiten in die folgenden Kategorien unterteilen:

- Textverarbeitungsprogramme (z.B. Microsoft Word, LibreOffice Writer)
- Wikis
- Visualisierungsprogramme (z.B. Microsoft Visio, Dia[1])
- Textbasierte Diagrammgeneratoren (z.B. LaTeX/TikZ[2], PlantUML[3])
- Modellierungsprogramme (z.B. Enterprise Architect[4], Modelio[5])

Textverarbeitungsprogramme sind hierbei die simpelsten Werkzeuge. Bekannte Vertreter dieser Kategorie sind Microsoft Word und LibreOffice Writer. Die Bedienung ist sehr intuitiv und die meisten Anwender sind im Umgang mit diesen Anwendungen bereits geübt. Die Möglichkeiten zur Diagrammerstellung sind allerdings eher spärlich. Hier ist es ratsam, für die Diagrammerstellung eine weitere Software, beispielsweise einem Visualisierungprogramm, zu verwenden.

[1] http://dia-installer.de/
[2] https://sourceforge.net/projects/pgf/
[3] http://plantuml.com/
[4] https://www.sparxsystems.de/
[5] https://www.modelio.org/

Eine andere simple Variante sind Wikis. Sie stehen den Textverarbeitungsprogrammen sehr nah und sind sehr flexibel was die Textgestaltung betrifft. Da viele Entwickler die Bedienung von Wikis bereits gewohnt sind entsteht auch hier kein nennenswerter Einarbeitungsaufwand. Wikis sind primär für die Gestaltung von Texten ausgelegt. Die meisten Wiki-Systeme bieten jedoch die Möglichkeit, den Funktionsumfang durch ein Plug-in-System zu erweitern. Diese können das Wiki um eine Diagrammerstellung erweitern.

Sowohl Vor- als auch Nachteil von Wikis ist, dass sie als Website betrieben werden. Somit wird immer der aktuelle Stand verwendet und die Gefahr veraltete Dokumente zu verwenden sinkt. Teilweise werden Änderungen sogar von den Wiki-Systemen automatisch archiviert und bleiben somit bei Bedarf verfügbar. Eine Website bedeutet jedoch auch, das die Dokumentation nur online einsehbar ist. Da üblicherweise keine lokale Kopie existiert kann es je nach Zugriffsrechten auf das Wiki zu Einschränkungen bei der Arbeit, beispielsweise während Kundenterminen, kommen.

Das Gegenstück zu Textverarbeitungsprogrammen sind Visualisierungsprogramme wie Microsoft Visio oder Dia. Diese sind auf Diagramme spezialisiert und ähnlich intuitiv bedienbar wie Textverarbeitungsprogramme. Sie verfügen jedoch nur über eingeschränkte Fähigkeiten zur textuellen Beschreibung. Kurze Erläuterungen können in die Diagramme integriert werden. Für umfassendere Dokumentationen sollten jedoch die Kombination mit einer textorientierten Anwendung in Betracht gezogen werden.

Eine weitere Kategorie sind Werkzeuge, die aus einer textuellen Beschreibung Diagramme erzeugen. Software wie LaTeX/TikZ oder PlantUML verwenden eine einfach Domain-specific Language (DSL) zur Beschreibung der Diagramme. Auch unterstützen viele Wikis eine solche DSL und können daher gut kombiniert werden. Da auf diese Weise erstellte Diagramme mit Textdateien beschrieben sind, können sie ohne Probleme zusammen mit dem Programmcode in den etablierten SCM-Systemen gesichert werden.

Als weitere Ausbaustufe von Visualisierungsprogramme können Modellierungsprogramme gesehen werden. Programme wie Enterprise Architect oder Modelio sind speziell für die Softwaremodellierung entwickelt. Im Gegensatz zu Visualisierungsprogrammen findet die Datenpflege hier nicht auf einzelnen Sichten statt. Die Software verwaltet eigene Instanzen aller Objekte und Komponenten. Architektursichten werden aus diesen Instanzen zusammengestellt und bleiben mit diesen verbunden. Dadurch ist es beispielsweise nicht nötig alle Sichten händisch zu aktualisieren. Die Änderungen werden am Datenbestand durchgeführt und können von der Software automatisch auf alle Sichten angewandt werden. Auch kann die Software hierdurch eine Navigation durch die verschiedene Verfeinerungsebenen anbieten.

Zusätzlich bieten einige Vertreter dieser Kategorie die Generierung von Code aus den Modellen und die Generierung von Modellen aus Programmcode an. Modellierungstools sind daher besonders gut zum Modellieren der Software aus den Architektursichten heraus geeignet. Die Bedienung solcher Tools ist jedoch entsprechend komplex und erfordert Erfahrung und Schulung.

Für die Dokumentation des NuPTools-Projektes wird ein System benötigt, mit dem jeder Entwickler Informationen über das Projekt erfahren und bei Bedarf erweitern kann. Da das Projekt vorrangig dokumentiert werden soll und keine Modellierung über Architektursichten geplant ist, kommen die Vorteile von Modellierungswerkzeuge nicht zum tragen. Es besteht daher kein Bedarf an Modellierungswerkzeugen.

Zum Einsatz kommt eine Kombination aus einem Wiki und einem Diagrammgenerator. Für das NuPTools-Projekt ist bereits ein Wiki-System im Betrieb. Dieses enthält bisher nur wenige Informationen. Es ist dennoch als Informationsquelle bekannt und die Weiterverwendung ist wünschenswert. Zur Diagrammerstellung wird das Wiki mit PlantUML kombiniert. Dies bietet den Vorteil, dass es aufgrund seiner einfachen Beschreibungssprache Teile der Architektur, wie in Abschnitt 5.3 beschrieben, generiert werden können.

5.3 Generierung von Architekturdokumentation

Für das Generieren von Architeturdokumentation im NuPTools-Projekt wird ein eigener Generator entwickelt. Dieser generiert PlantUML-Dateien. PlantUML hat einige Eigenschaften, die für das Generieren vorteilhaft sind:

- Diagrammedefinitionen können auf mehrere Dateien verteilt werden
- Fehlende Elemente werden automatisch aus vollqualifizierenden Namen erstellt
- Klassen werden entsprechend ihrem vollqualifizierendem Namen in Namespaces angeordnet
- Elemente sind über ihren vollqualifizierenden Namen adressierbar
- Pakete werden nicht automatisch geschachtelt

Die Verteilung der Diagramme über verschiedene Dateien hat besonders bei der Generierung Vorteile. Auf diese Weise können generierte Daten und händisch erstellte Informationen kombiniert werden, indem die generierten Dateien in händisch erstellte bzw. zusammengestellte Diagramme eingebunden werden.

```
class Package.Class
```

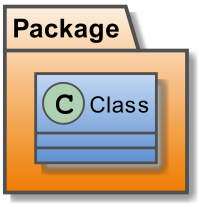

Abbildung 5.5: Beispiel eines Diagrammes mit implizit erzeugtem Paket

Durch das automatische Erzeugen von Elementen müssen nicht alle Elemente definiert werden. Dies wird für Kombinationen verschiedener Informationen eingesetzt. Wie in Abbildung 5.5 zu sehen ist, wurde das Paketelement ohne explizite Definition erzeugt.

Bei der Verwendung von Namespaces ist PlantUML in der Lage, Klassen entsprechend ihrem vollqualifizierendem Namen in den Namespaces anzuordnen. Beispielhaft ist dies in Abbildung 5.6 zu sehen. Klassen müssen nicht innerhalb der Verschachtlungsstruktur eingebettet werden.

Ebenfalls in Abbildung 5.6 zu sehen ist eine Notiz, die über den vollqualifizierenden Namen der Klasse an diese adressiert wird. Dies kommt besonders bei der Kombination von Informationen aus verschiedenen Dateien zum tragen, da die einzelnen Informationen unabhängig voneinander definiert werden können.

Im Gegensatz zu Klassen werden Pakete und Namespaces nicht automatisch ineinander angeordnet. Wie in Abbildung 5.7 auf der nächsten Seite zu sehen können Namespaces, die Teil eines anderen Namespaces sind, neben diesem stehen. Der Generator nutzt dieses Verhalten, um Abhängigkeiten zwischen den Paketen zu zeigen.

```
namespace Ns1{
  namespace Ns2{
  }
}
class Ns1.Ns2.Class
note right of Ns1.Ns2.Class:
   Comment
```

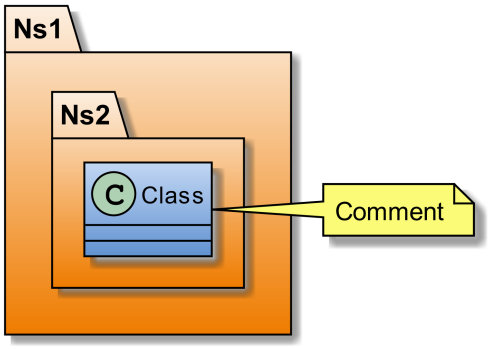

Abbildung 5.6: Beispiel eines Diagrammes mit automatischen angeordneten Klassen und Notizen

```
namespace Ns1{
}
namespace Ns1.Ns2{
}
Ns1 <-left- Ns1.Ns2
```

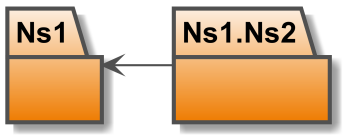

Abbildung 5.7: Beispielhafte Abhängigkeiten zwischen Paketen

Der entwickelte Generator erstellt drei Arten von Dokumenten:

- Auflistung alle Klassen in einer Komponente
- Auflistung aller Namensbereiche in einer Komponente
- Abhängigkeiten zwischen spezifizierten Komponenten

Diese Dokumente können je nach Bedarf miteinander oder mit händisch erstellten Informationen kombiniert werden. Beispiel hierfür ist die Darstellung einer Paketstruktur mit oder ohne Abhängigkeiten zwischen den Paketen.

5.4 SOLID-Prinzipien

Die SOLID-Prinzipien gelten als Leitfaden für gute Softwarearchitekturen. Starke (2015) erklärt die fünf Prinzipien an einfachen Beispielen. Sie lauten:

S Single-Responsibility-Prinzip
O Open-Closed-Prinzip
L Liskov-Substitution-Prinzip
I Interface-Segregation-Prinzip
D Dependency-Injection-Prinzip

Die Prinzipien können helfen, dass eine Software strukturiert und übersichtlich entworfen wird. Außerdem fördern die Prinzipien eine lose Kopplung der Komponenten und machen die Software leicht kontrollierbar.

5.4.1 Single-Responsibility-Prinzip

Nach dem Single-Responsibility- oder auch Separation-of-Concerncs-Prinzip sollen die Komponenten eines Systems immer genau eine spezifische Aufgabe haben. Beispielsweise sollte eine Logging-Komponente keine Konfigurationsdateien verwalten.

Eine Komponente, die mehrere Aufgaben erfüllt, kann nur durch eine Komponente mit den gleichen Fähigkeiten oder mehrere Komponenten ausgetauscht werden. Außerdem können Komponenten, die mehrere Aufgaben erfüllen, dazu neigen weitere Komponenten zur Erfüllung ihrer Aufgaben zu nutzen. Beispielsweise könnte eine Logger-Komponente, die selbstständig ihre eigene Konfigurationdateien verwaltet, diese Aufgabe an eine spezialisierte Komponente weiterreichen. Dies würde nicht nur eine unerwünschte Abhängigkeit zu der Konfigurationsverwaltung erzeugen. Es könnte auch Konflikten mit der regulär eingeplanten Konfigurationsverwaltung hervorrufen.

Das einhalten dieses Prinzips kann daher helfen, das System wartbarer zu gestalten. Außerdem hilft es sicherzustellen, das Komponenten lose gekoppelt sind.

5.4.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip besagt, dass Komponenten offen für Erweiterungen, aber geschlossen für Änderungen sein sollen. Die Komponente soll also nicht durch Codeänderungen erweitert werden.

Praktisch wird dies mithilfe der Polymorphie umgesetzt. Für Komponenten, die verschiedene Verhalten aufweisen müssen, werden Schnittstellen definiert. Für jedes Verhalten kann ein spezialisierter Typ diese Schnittstelle implementieren. Jeder dieser spezialisierten Typen unterscheidet sich in seinem Verhalten, aber alle Typen sind gleich zu verwenden. Zum Erweitern der Funktionalität kann ein weiterer spezieller Typ implementiert werden, ohne das Änderungen an dem anwendenden Code vollzogen werden müssen.

5.4.3 Liskov-Substitution-Prinzip

Nach dem Liskov-Substitution-Prinzip müssen Basisklassen durch die von ihnen abgeleiteten Klassen ersetzbar sein. Das Verhalten der abgeleiteten Klasse darf sich nicht so ändern, dass anwendende Programme beschädigt werden. Das bedeutet, dass beispielsweise eine Klasse, deren Basisklasse mehrere Parameter berechnet, die Parameter zwar anders, aber nicht weniger Parameter berechnen darf. Ein Anwender dieser Klasse muss

erwarten können, dass die Parameter, die er auf Grundlage der Basisklasse verwendet, auch in der abgeleiteten Klasse verfügbar sind.

5.4.4 Interface-Segregation-Prinzip

Um Abhängigkeiten auf unnötige Funktionalitäten zu vermeiden, fordert das Interface-Segregation-Prinzip, dass jede Schnittstelle nur eine Aufgabe beschreibt. Nach einer Änderung an einer Schnittstelle bedarf jede Implementierung dieser Schnittstelle eine Aktualisierung. Bei Schnittstellen, die mehrere Aufgaben vereinen, müssen auch Implementierungen aktualisiert werden die zwar die Schnittstelle, aber nicht die geänderte Aufgabe bedienen. Bei Schnittstellen, die entsprechend dem Interface-Segregation-Prinzip nur für eine Aufgabe entwickelt wurden, bedienen alle Implementierungen diese Aufgabe. Somit müssen keine unnötigen Aktualisierungen durchgeführt werden.

5.4.5 Dependency-Injection-Prinzip

Dem Dependency-Injection-Prinzip folgend instanziieren Anwenderklassen ihre Abhängigkeiten nicht eigenständig. Sie sollen stattdessen nur auf den entsprechenden Schnittstellen arbeiten. Hierdurch ist die Anwenderklasse nicht von einer speziellen Implementierung der Schnittstelle abhängig. Die Kontrolle über die Instanziierung der Abhängigkeiten verlagert sich dadurch aus der Klasse heraus. Die Instanziierung kann durch Dependency Injection an einer zentralen Stelle kontrolliert werden und bestimmen welche spezielle Implementierung zum Einsatz kommt. Ebenso kann hier kontrolliert werden, ob ein Typ mehrfach oder einmalig für alle verwendenden Komponenten instanziiert wird.

5.5 Analyse der bestehenden Architektur

In Abbildung 5.8 auf der nächsten Seite ist die Bausteinsicht der bestehenden Architektur zu sehen. Eine Übersicht über die Komponenten bietet Tabelle 5.1 auf Seite 63.

In dieser Architektur besteht das Plug-in aus einem *AddInServer*, der den Einstiegspunkt für die Hostanwendung darstellt. Diese Komponente enthält neben der Logik zur Integration in die Hostanwendung auch den größten Teil der funktionalen Logik. Zur Kommunikation mit der Hostanwendung wird das NuPHostFramework verwendet. Hiervon werden wichtige Teile bereits in den Plug-in-Code integriert. Der hier angewandte Stil

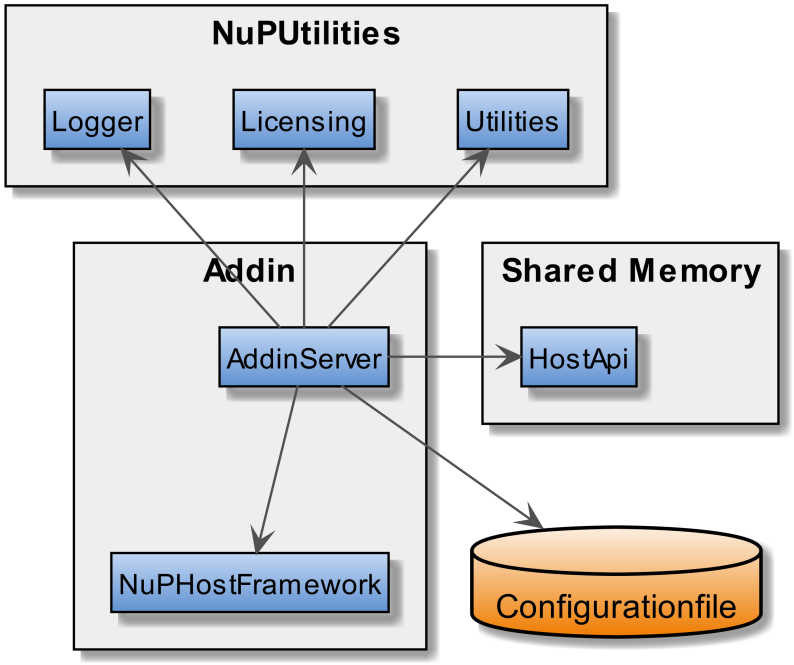

Abbildung 5.8: Bausteinsicht der bestehenden Architektur

birgt zwei Probleme. Generisch verwendbare Programmteile werden in jedes Plug-in integriert. Sie sind somit potentiell zur Laufzeit mehrfach vorhanden. Das zweite Problem ist, dass die Plug-ins sehr unübersichtlich werden. Die *AddinServer*-Klassen beinhalten fast sämtlichen Code und werden dadurch extrem groß. Auch ist es hierdurch schwer zu unterscheiden, auf welchen Programmteil bestimmte Codestücke Einfluss haben. Also beispielsweise, ob eine Methode Teil der Integrationslogik ist, eine Produktfunktionalität umsetzt oder Konfigurationsdaten verwaltet.

Generische Komponenten des NuPTools-Projektes sind Teil der NuPUtilities. Nur wenige Teile der NuPUtilities sind Modular entworfen. Der größte Teil, hier zu Utilities zusammengefasst, lässt sich als Hilfsmethoden beschreiben. Die NuPUtilities werden von der AddinServer-Komponente verwendet. Dabei wird auf eine gemeinsame Nutzung der Instanzen durch verschiedene Plug-ins verzichtet. Somit entsteht je installiertem Plug-in zur Laufzeit mindestens eine Instanz der verwendeten Komponenten.

Zur Konfiguration der Plug-ins wird in einigen Plug-ins eine Konfigurationsdatei von der AddinServer-Komponente verwaltet. Bis auf kleinere Hilfsmethoden aus den NuPUtilities ist die gesamte Verwaltung dieser Konfigurationen Aufgabe der AddinServer-Komponente. Diese Situation birgt verschiedene Problemen. Die zusätzliche Aufgabe der Server-Komponente vergrößert diese und nimmt somit negativen Einfluss auf die Wart-

barkeit. Zusätzlich erzeugt die Verwaltung der Konfiguration durch das Plug-in für jedes Plug-in wiederholenden Entwicklungsaufwand. Ein weiteres Problem ist, dass diese Art der Konfigurationsverwaltung viel Spielraum für inkonsequente Methodik lässt. Dieser wird Teils sehr weit ausgenutzt, sodass selbst erfahrene Entwickler häufig keinen Überblick über die Konfiguration haben.

Der Zugriff auf die Hostanwendung erfolgt über die HostApi. Diese wird von der Hostanwendung verwaltet und von dem Plug-in je nach Bedarf verwendet.

Komponente	**Beschreibung**
AddinServer	Die AddinServer-Komponente ist der Einstiegspunkt in das Plug-in. Sie deklariert die zur Hostanwendung hinzuzufügenden Elemente und Funktionen sowie die meißten Plug-in-funktionalitäten.
NuPHostFramework	Einige Hilfsfunktionen zur Kommunikation mit der jeweiligen Hostanwendung sind in der NuPHostFramework-Komponente zu finden. Diese Funktionen sind teilweise stark mit den Plug-ins gekoppelt. Aktualisierungen an dieser Komponente erzeugen dadurch so großen Anpassungsaufwand, dass eine Aktualisierung fast unmöglich ist.
NuPUtilities (Logger, Licensing, Utilities)	Die NuPUtilities sind eine Sammlung von gemeinsam genutzen Komponenten und Hilfsfunktionen. Hierzu zählen einige Komponenten wie ein Logger und eine Lizensierungskomponente. Der größte Teil sind die zu Utilities zusammengefassten Hilfsmethoden.
Configurationfile	Die bestehenden Plug-ins verwalten Ihre Konfigurationsdateien selbstständig. Daher muss für jedes Plug-in eine eigene Konfigurationsverwaltung entwickelt werden. Aufgrund des großen Aufwandes hierfür wurde diese i.d.R. sehr knapp gehalten, sodass Konfigurationsänderungen händisch in einer XML-Datei vorgenommen werden müssen und einen Neustart der Hostanwendung erfordern.
HostApi	Die HostApi ist die Schnittstelle zur Hostanwendung. Sie wird benötigt, um Zugriff auf Daten aus der Hostanwendung zu erhalten.

Komponente	Beschreibung

Tabelle 5.1: Übersicht über die Komponenten der bestehenden Architektur

5.6 Überarbeitete Architektur

Bei der Überarbeitung der Architektur wird besonders auf die SOLID Prinzipien Wert gelegt. Hierdurch wird die Wartbarkeit und die Entkopplung der Komponenten verbessert. Außerdem wird die gemeinsame Verwendung von Komponenten durch verschiedene Plug-ins verbessert.

Abbildung 5.9 auf der nächsten Seite zeigt die Bausteinsicht der überarbeiteten Architektur. Tabelle 5.2 auf Seite 66 gibt eine Beschreibung der einzelnen Komponenten. Die Architektur ist in die drei Bereiche „Shared Memory“, „Addin“ und „NupFramework“ unterteilt. Im Vergleich zu der bestehenden Architektur fällt hier besonders die deutlich größere Anzahl an Komponenten im Shared Memory auf. Des Weiteren zeigt das Diagramm auch eine deutlich geringere Verantwortung der AddinServer-Komponente. Auch die bisherigen NuPUtilities sind als NuPFramework neu strukturiert. Viele der im Shared Memory befindlichen Komponenten gehören organisatorisch ebenfalls zum NuPFramework. Sie sind jedoch technisch losgelöst und nicht mehr fest mit allen anderen Framework.Komponenten gekoppelt.

Die AddinServer-Komponente dient nun nur der Integration in die Hostanwendung. Sämtliche Plug-in-Logik ist in eine separate Komponente ausgelagert. Die AddinServer-Komponente wird hierdurch deutlich verschlankt und übersichtlicher. Einige allgemeine Hilfsmethoden und Hilfsmethoden zur Kommunikation mit der Hostanwendung werden durch das NuPFramework bereitgestellt. Sowohl der AddinServer als auch die Plug-in-Logik haben direkten Zugriff auf das Framework.

Für die Instanziierung und Zuordnung der verwendeten Komponenten verwendet der AddinServer einen Dependency-Injection-Container. Dieser wird von allen Plug-ins geteilt und dient somit als zentrale Kontrolleinheit für alle NuPTools. Er übernimmt die Verwaltung der Komponenten wie beispielsweise dem Updater, der Lizensierung und dem Logger. Abschnitt 5.9 auf Seite 70 geht vertiefend auf den Dependency-Injection-Container ein.

Eine neue Komponente im Vergleich zu der bestehenden Architektur ist der Messenger. Der in Abschnitt 5.11 auf Seite 76 detaillierter beschriebene Messenger dient der Kom-

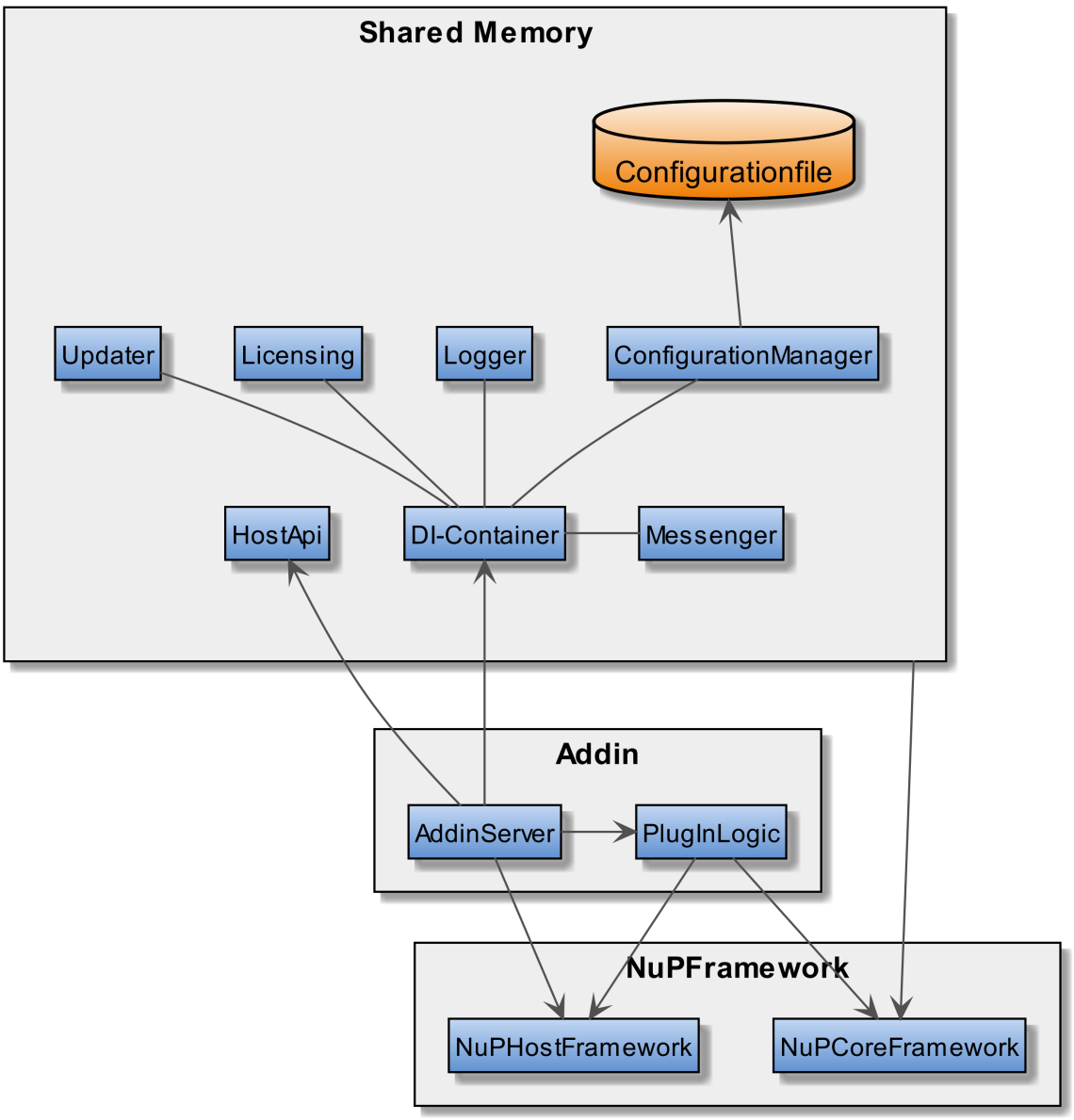

Abbildung 5.9: Entwurf der überarbeiteten Architektur

munikation zwischen Komponenten. Komponenten können über den Messenger Benachrichtigungsereignisse auslösen und empfangen.

Eine weitere neue Komponente ist der Konfigurationsmanager. Der Konfigurationsmanager verwaltet alle Konfigurationsdateien. Durch ihn müssen Plug-ins keine eigene Konfigurationsverwaltung implementieren. Sämtliche Details der Verwaltung werden von dem Manager übernommen. Hier kommt auch der Messenger zum Einsatz. Über ihn werden beispielsweise Mitteilungen über geänderte Konfigurationen gesendet. So können verwen-

dende Komponenten darüber informiert werden, dass ihre verwendete Konfiguration nicht mehr gültig ist. Im Detail wird der Konfigurationsmanager in Abschnitt 5.12 auf Seite 77 erläutert.

Komponente	**Beschreibung**
AddinServer	Die AddinServer-Komponente ist der Einstiegspunkt in das Plug-in. Sie deklariert die zur Hostanwendung hinzuzufügenden Elemente und Funktionen. Die Ausführung der Funktionen wird durch die AddinServer-Komponente an die Plug-in-Logik delegiert.
PlugInLogic	Die Plug-in-Logik beinhaltet alle Funktionalitäten des jeweilige Plug-ins.
NuPCoreFramework	Das NuPCoreFramework beinhaltet allgemeine Hilfsmethoden und Erweiterungen.
NuPHostFramework	Einige Hilfsfunktionen, zur Kommunikation mit der jeweiligen Hostanwendung sind in der NuPHostFramework-Komponente zu finden.
Updater	Die Updater-Komponente dient zur Softwareaktualisierung. Sie prüft im Hintergrund ob neue Softwareversionen verfügbar sind. Falls Aktualisierungen möglich sind, lädt der Updater die Installationsdaten im Hintergrund und benachrichtigt den Benutzer.
Licensing	Das Lizenzierungssystem überprüft die Softwarelizenz. Falls keine gültige Lizenz gefunden wird, fordert das System den Benutzer zur Eingabe gültiger Lizenzinformationen auf oder blockiert das Laden und Integrieren des betroffenen Plug-ins in die Hostanwendung.
Logger	Der Logger dient zum Protokollieren von Systemereignissen.
Configuration-Manager & Configurationfile	Der Konfigurationsmanager verwaltet alle Konfigurationsdateien. Dazu lädt und speichert er die Konfigurationsdateien selbstständig. Das Zusammenführen von Konfigurationen verschiedener Plug-ins in eine Datei wird durch den Manager ermöglicht. Er bietet eine einfach Schnittstelle zum Beziehen eines Konfigurationstyps an.

Komponente	Beschreibung
Messenger	Der Messenger dient zur Kommunikation zwischen Komponenten. Über den Messenger können Komponenten ungerichtete Nachrichten senden. Andere Komponenten können sich bei ihm für spezifische oder unspezifische Benachrichtigungen registrieren. Er ermöglicht die Kommunikation zwischen lose gekoppelten Komponenten.
HostApi	Die HostApi ist die Schnittstelle zur Hostanwendung. Sie wird benötigt, um Zugriff auf Daten aus der Hostanwendung zu erhalten.

Tabelle 5.2: Übersicht über die Komponenten der überarbeiteten Architektur

5.7 Strukturbereinigung

Zur besseren Orientierung innerhalb des NuPFrameworks wird die Namespaces-Struktur neu organisiert. Microsoft zitiert hierzu, wie bei den in Abschnitt 3.6.1 auf Seite 34 beschriebenen Stilregeln auch, das Regelwerk von Cwalina und Abrams (2008). Diese empfehlen das in Ausschnitt 5.1 gezeigt Schema.

Der erste Abschnitt sollte ein Firmennamen sein, um Überschneidungen mit anderen Firmen oder Entwicklern zu vermeiden. Der nächste Abschnitt sollte eine Produkt- oder Technologiebezeichnung sein. Darauf folgt eine Bezeichnung für die Funktionalität der in dem Namespace beinhalteten Typen. Als letzten Abschnitt können nötige Unterteilungen frei gewählt werden. Dieses Schema wird im NuPFramework als Richtlinie festgelegt. So entstehen Namespaces wie beispielsweise *Nupis.Core.Injection* oder *Nupis.Core.ConsoleLogger*.

Neben dem Schema nennt Cwalina und Abrams (2008) weitere Regeln. Diese spezifizieren die Vorgaben weiter. Die folgenden Regeln werden von Microsoft Corporation (2017) zitiert:

- Beginne Namespace-Namen mit einem Firmennamen.

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

Ausschnitt 5.1: Namespace-Schema nach Cwalina und Abrams (2008)

- Verwende beständige, versionsunabhängige Produktnamen im zweiten Abschnitt des Namspaces.
- Verwende keine organisatorische Hierarchien für Namespace-Hierachien. Also beispielsweise keine Abteilungsnamen, weil diese oft kurzlebig sind.
- Verwende die Pascal Case-Notation und trenne Namespace-Bereiche mit Punkten. Verwende die Schreibweise von Marken, auch wenn diese von der normalen Schreibweise abweicht.
- Wähle Begriffe in der Mehrzahl wenn dies sinnvoll ist.
- Verwende nicht den gleichen Namen für einen Namespace und einen Typen in dem Namespace.
- Verwende keine allgemeinen Begriffe für Typen wie beispielsweise *Element*, *Node*, *Log* und *Message*.
- Gib den selben Namen nicht mehreren Typen innerhalb eines Produkt-Namespaces.
- Verwende keine Typennamen, die bereits in Kern-Namespaces verwendet werden. Dies umfasst alle Namespaces, die unterhalb des System-Namespace angeordnet sind. Beispielsweise *System.IO* und *System.Xml*.
- Ein Technologie-Namespace beinhaltet alle Typen und Namespaces, die mit den beiden gleichen Namen beginnen. Beispielsweise *Microsoft.Build.Utilities* und *Microsoft.Build.Tasks*. Weise keine Typnamen zu, die einen Konflikt mit einem anderen Typen in dem selben Technologie-Namespace verursachen.
- Weise keine Typnamen zu, die einen Konflikt zwischen einem Technologie-Namespace und einem Anwendungs-Namespace verursachen.

Die bereinigte Struktur wird durch Abbildung 5.10 auf der nächsten Seite und Tabelle 5.3 auf Seite 69 beschrieben. Abbildung 5.10 zeigt die aktuelle Übersicht über die Komponenten des NuPFrameworks. Hier werden die Funktionen der Komponenten kurz erklärt. Das Diagramm soll den Entwickler ermöglichen, schnell die nötigen Komponenten ausfindig zu machen. Auch bei der Weiterentwicklung des NuPFrameworks hilft dieses Diagramm, indem es eine Übersicht über die aktuellen Komponenten gibt.

Für die detaillierteren Namespaces bietet Tabelle 5.3 auf Seite 69 eine Richtlinie. In jeder Komponente sollten die Typen in Namespaces mit den gezeigten Namen angeordnet

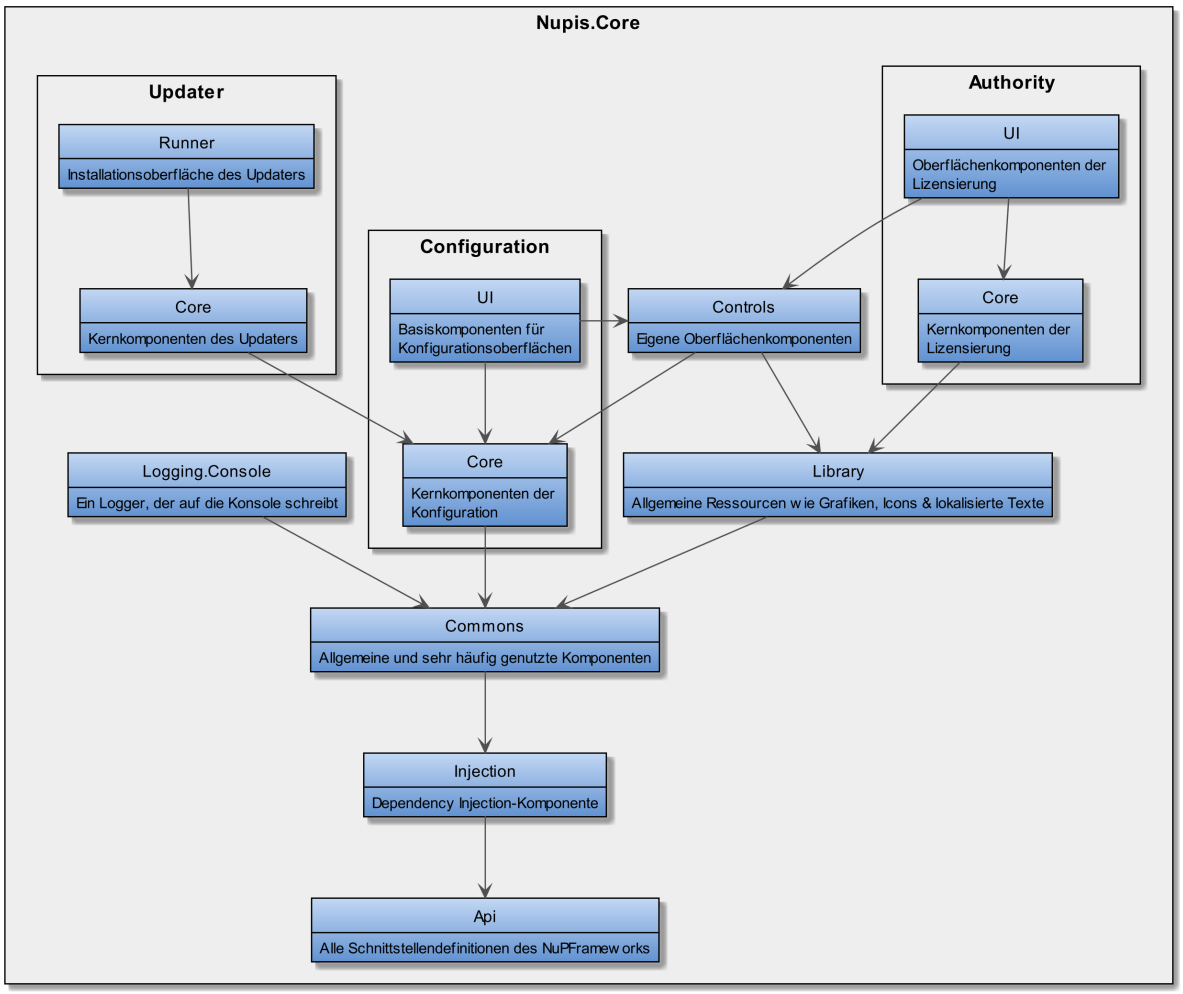

Abbildung 5.10: Beschreibung der Komponenten des NuPFrameworks sowie deren Abhängigkeiten untereinander.

werden. Durch diese Richtlinie soll eine einheitliche Struktur entstehen. Jeder Entwickler kann somit abschätzen, wo bestehende Typen untergebracht sein sollten. Aufgrund geplanter Veröffentlichungstermine werden viele der Änderungen zur Anpassung an die neuen Richtlinien erst zur nächsten Version umgesetzt.

Namespace	Beschreibung
Data	Alle Datenklassen dieser Komponente.
Enumerations	Alle Enumerationen dieser Komponente.
PlugIn	Alle Klassen, die zur Definition des PlugIns benötigt werden.
Types	Alle Hilfstypen und Komponentenlogiktypen.
ViewModels	Alle Views dieser Komponente.
Views	Alle ViewModels dieser Komponente.

Tabelle 5.3: Richtlinie für Namespaces im NuPFramework

5.8 Abhängigkeiten im NuPFramework

Die Abbildung 5.10 auf Seite 68 zeigt die Abhängigkeiten innerhalb des NuPFrameworks. Zur besseren Übersicht wurden in dem Diagramm direkte Abhängigkeiten, die auch als indirekte Abhängigkeiten bestehen, entfernt. Da die überarbeitete Architektur Wert auf lose gekoppelte Komponenten legt, sind hier bereits Ansätze der losen Kopplung zu erkennen. Weitere Optimierungen der Abhängigkeiten werden, wie bei der Strukturbereinigung auch, in der nächsten Version umgesetzt. Diese Optimierungen umfassen die folgenden Änderungen.

- Die Nupis.Core.Commons-Komponente ist in der aktuellen Version von Nupis.Core.Injection abhängig. Dies ist durch das in Abschnitt 5.9.3 auf Seite 73 beschriebene Service Locator-Muster bedingt. Eine weitere Nutzung der Abhängigkeit ist durch eine nicht vollständig entkoppelte Schnittstelle gegeben.
- Die Abhängigkeit zwischen Nupis.Core.Authority.Core und Nupis.Core.Library entsteht durch die Verwendung einer einzelnen Textressource. Hier muss geprüft werden, ob die Ressource sinnvoll verlegt werden kann. Die Verlegung in die Nupis.Core.Authority.Core kann eine Option sein.
- Die Typen des Konfigurationssystems sind in der aktuellen Version noch stark gekoppelt. Daher erzeugt im Moment jedes Konfigurationsobjekt eine Abhängigkeit auf die Nupis.Core.Configuration.Core-Komponente. Hier müssen viele Schnittstellen überarbeitet und die Abhängigkeit gelöst bzw. auf die API-Komponente verlegt werden.

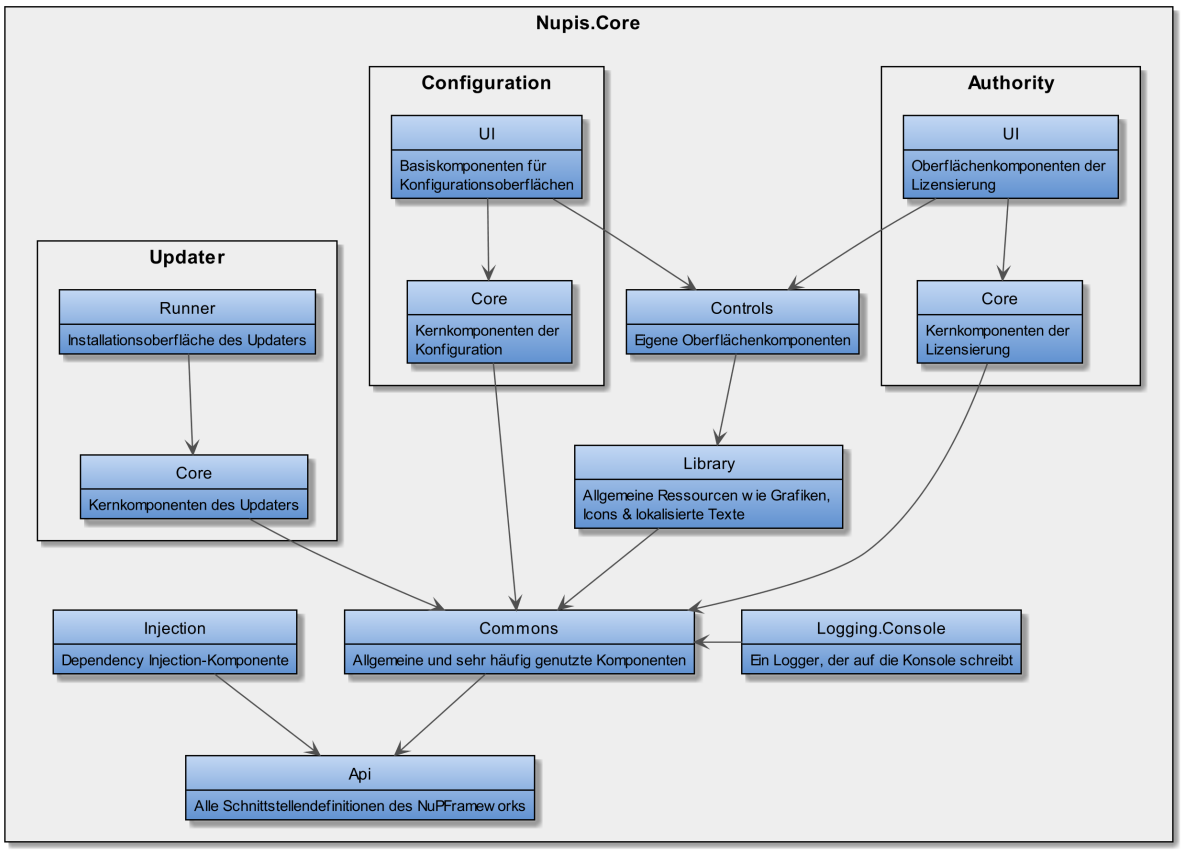

Abbildung 5.11: Übersicht über die zukünftigen Abhängigkeiten innerhalb des NuPFrameworks

Das Ergebnis dieser Änderungen zeigt Abbildung 5.11. Nupis.Core.Api bleibt der zentrale Knoten des NuPFrameworks. Auch Nupis.Core.Commons wird noch von vielen Komponenten verwendet, da diese Komponente viele allgemeine Hilfsmethoden zur Verfügung stellt.

Von Nupis.Core.Library sind nur noch Komponenten mit grafischen Oberflächen oder Steuerelementen abhängig. Außerdem sind nur erweiternde Komponenten wie Nupis.Core.Configuration.UI von den jeweiligen Kernkomponenten abhängig. Die verbleibenden Abhängigkeiten bringen somit keine unnötigen Abhängigkeiten mit sich.

5.9 Dependency Injection

Dependency Injection ist ein Entwurfsmuster zur losen Kopplung von Komponenten. Außerdem ermöglicht das Muster eine bessere Kontrolle über die verwendeten Komponenten.

Dazu werden die verwendeten Komponenten an einer zentrale Stelle im Code erstellt und zugewiesen. Hall (2015) erklärt das Muster an Beispielen in C#.

5.9.1 Grundlagen von Dependency Injection

Ziel der Dependency Injection ist es, die feste Kopplung zwischen Komponenten zu lösen. Mit Kopplung ist die Bindung an einen speziellen Typ gemeint. Die als Abhängigkeit bezeichneten Komponenten werden bei der Dependency Injection durch Schnittstellen ersetzt. Eine Komponente verwendet somit keine spezielle Implementierung sondern nur eine abstrakte Schnittstelle. Zusätzlich wird die spezifische Komponente nicht von der verwendenden Komponente instanziiert. Die Instanziierung übernimmt eine zentrale Kontrollstelle, welche die verwendete Komponente anschließend in die verwendende Komponente injiziert. Auf diese Weise wird die Kopplung zwischen Komponenten gelöst. Zusätzlich wird die Zusammenstellung der Komponenten bzw. Instanzen von der zentralen Stelle delegiert.

Ein Beispiel für die Anwendung verschiedener Komponenten ist eine Datenbankschnittstelle. Mit Hilfe von Dependency Injection kann die spezielle Implementierung für einen bestimmten Datenbanktyp ausgewählt werden. Die anwendende Komponente arbeitet dabei ausschließlich gegen eine definierte Schnittstelle. Welche Implementierung dieser Schnittstelle zur Anwendung kommt wird von der zentralen Kontrollstelle der Dependency Injection ausgewählt.

Die Injizierung der Abhängigkeiten kann über drei Arten erfolgen.

- Eine Art ist die Property-Injection. Hierbei wird die Abhängigkeit über Klasseneigenschaften in den Typ eingebunden. Dieses Verfahren bietet den Vorteil, dass die verwendete Instanz zur Laufzeit ausgetauscht werden kann. Beim Austauschen der Instanz ist jedoch Vorsicht geboten, da Instanzen auch von anderen Komponenten verwendet werden können. Der Austausch von Abhängigkeiten kann daher unerwartet Einfluss auf andere Abläufe nehmen.

- Eine besseres Verfahren zum bedarfsabhängigen Injizieren ist die Method-Injection. Hierbei wird die Abhängigkeit beim Methodenaufruf übergeben. Dieses Verfahren eignet sich besonders, wenn die Abhängigkeit von dem aufrufenden Kontext abhängig ist und nur von einer oder wenigen Methoden benötigt wird. Als Nachteil kann hier gesehen werden, dass das aufrufende Objekt die Abhängigkeit kennen muss.

- Die dritte Möglichkeit ist die Constructor-Injection. Hierbei werden die Abhängigkeiten an den Konstruktor übergeben. Bei diesem Verfahren werden die Abhängigkeiten in der Regel nicht mehr gewechselt. Dies stellt das Verfahren mit dem geringsten Aufwand und der höchsten Sicherheit dar. Durch die Konstruktorinjektion kann der Anwender in der Regel davon ausgehen, dass alle benötigten Abhängigkeiten bereits injiziert wurden und keine verhaltensändernden Auswechslungen vorgenommen werden.

In der Praxis wird im .Net-Bereich vorwiegend die Constructor-Injection angewandt. Von der Verwendung der Property-Injection wird häufig abgeraten. Simple Injector Contributors (2017) nennt dazu verschiedene Begründungen. Als implizite Injektion, also der automatischen Injektion, müssten hierbei nicht auflösbare Abhängigkeiten ignoriert werden. Die Logik der Klasse kann somit nicht mehr davon ausgehen, dass alle Abhängigkeiten injiziert wurden. Bei der expliziten Property-Injection muss die zu injizierende Klasse eine Referenz auf die Dependency Injection-Bibliothek haben, um die zu injizierenden Eigenschaften zu markieren. Um diese Referenz zu vermeiden wird auch von der expliziten Injektion abgeraten.

Auch für das Auflösen der Abhängigkeiten, also das Auswählen der zu injizierenden Instanzen, kann auf verschiedene Weisen geschehen. Hall (2015) nennt hierfür die folgenden Verfahren:

- Poor-Man's-Dependency-Injection: Bei diesem Verfahren werden die Instanzen aller Abhängigkeiten zum Programmstart vom Entwickler händisch erstellt und injiziert. Dieses Verfahren ist sehr einfach, kann jedoch schon bei kleinen Projekten viel Code erzeugen. Zusätzlich birgt dieses Verfahren den Nachteil, dass das Erstellen aller Objekte zum Programmstart den Start verzögern kann.

- Manuelle Registrierung: Bei der manuellen Registrierung wird ein Dependency-Injection-Container verwendet. In diesem werden die Schnittstellen und die aufzulösenden Typen registriert. Alternativ kann zu Schnittstellen auch eine Fabrikmethode, also eine Methode zur Erstellung komplexer Objekte, registriert werden. Zum injizieren der Abhängigkeiten erstellt der Container Instanzen der registrierten Typen.

- Konvention über Konfiguration: Mit Konvention über Konfiguration wird ein Verfahren beschrieben, das anstelle einer Registrierung Konventionen zum Auflösen der Typen verwendet. Beispielsweise kann eine Regel definiert werden, die zu jedem

Oberflächentyp mit dem Suffix „View“ automatisch ein ViewModel mit dem selben Namen und dem Suffix „ViewModel“ auswählen.

Empfehlenswert ist eine Kombination aus der manuellen Registrierung und Konventionen. Viele Typen können nach einem einfachen Schema erkannt werden. Durch Konventionen kann daher viel Konfigurationsaufwand vermieden werden. Für Typen, die nicht einfach durch ein Schema erkennbar oder auswählbar sind, kann dennoch mit der manuellen Registrierung ein anzuwendender Typ konfiguriert werden.

5.9.2 Dependency-Injection-Systeme in .Net

Für .Net-Anwendungen sind unzählige Dependency-Injection-Systeme verfügbar. Diese bieten in der Regel einen Container an, der die Verwaltung und Injektion umsetzt. Für die Benutzung muss der Entwickler nur die benötigten Typen registrieren und mit Hilfe des Containers auflösen.

Palme (2017) testet und vergleicht eine große Anzahl an Dependency-Injection-Containern. In dem Test wird im Besonderen die Geschwindigkeit betrachtet. Zusätzlich wird auch der generelle Funktionsumfang berücksichtigt. Basierend auf dem Vergleich empfiehlt Palme (2017) drei Container. Diese bieten sowohl eine gute Leistung als auch einen guten Funktionsumfang. Die empfohlenen Container sind DryIoc[6], LightInject[7] und SimpleInjector[8].

Für das NuPTools-Projekt wird der DryIoc-Container verwendet. Dieser bietet einen besonders großen Funktionsumfang. Die Plug-ins des NuPTools-Projektes werden von der Hostanwendung nacheinander geladen und installiert. Daher müssen Registrierungen auch nach der Installation eines vorangegangenen Plug-ins möglich sein. Dies ist mit dem DryIoC-Container möglich. Viele andere Container, wie beispielsweise der SimpleInjector-Container, verschließen sich nach dem Auflösen des ersten Typs.

5.9.3 Umsetzung der Dependency Injection

Für die Anwendung des Containers wird in dem NuPTools-Projekt eine Fassade implementiert. Diese beschränkt den Funktionsumfang auf die benötigten Funktionen und vereinfacht somit die Anwendung. Bei der Initialisierung der Fassade registriert sich diese für

[6] https://bitbucket.org/dadhi/dryioc
[7] https://github.com/seesharper/LightInject
[8] https://simpleinjector.org/index.html

AssemblyLoaded-Ereignisse. Sie wird somit beim Nachladen von Assemblies informiert. Jedes bereits geladenen oder nachgeladene Assembly wird von der Dependency Injection verarbeitet. Dabei werden die Typen der Assembly entsprechend festgelegter Regeln registriert.

Bei der Verarbeitung der Assemblies werden firmeneigene Assemblies durch ihren Namespace identifiziert. Alle anderen Komponenten, beispielsweise Komponenten des .Net-Frameworks, werden ignoriert. Die Initialisierung sucht während dieses Vorgangs nach bestimmten Typen für die Regeln definiert wurden und registriert diese. Es kommt somit das Konvention-vor-Konfiguration-Verfahren zur Anwendung. Nicht alle Situationen können durch allgemeingültige Regeln abgedeckt werden. Eine der Regel sucht daher nach Typen, die ein spezielles Interface implementieren. Dieses ermöglicht manuelle Registrierungen. Durch Kombination der beiden Systeme können die meisten Registrierungen automatisiert vorgenommen werden. Nur wenige Registrierungen müssen von einem Entwickler händisch programmiert werden.

Die Fassade des Dependency Injection-Containers implementiert zusätzlich das Service-Locator-Muster. Dieses im Allgemeinen nachteilige Muster wird von Hall (2015) erläutert. Es ermöglicht den Zugriff auf den Container von jeder Codestelle aus. Idealerweise erfolgt der Zugriff jedoch nur von einem einzelnen Einstiegspunkt aus. Von diesem Wurzelpunkt sollten die Laufzeitstruktur erstellt und kontrolliert werden.

Das Service Locator-Muster kann drei Probleme mit sich bringen. Durch die Verwendung eines Service Locator entstehen eine Abhängigkeiten zu dem Dependency Injection Container bzw. dessen Fassade. Bei entsprechend ausgeprägter Verwendung kann jede Komponente eine unerwünschte Abhängigkeit auf die Injektionsmechanismen erhalten. Das zweite Problem sind die versteckten Abhängigkeiten. Da der Anwender der Komponenten keine Abhängigkeiten bereitstellen muss werden die genutzten Abhängigkeiten durch die Verwendung eines Service Locators versteckt. Ein weiteres Problem entsteht beim Testen der Komponenten. Durch den Service Locator bezieht die Komponente ihre Abhängigkeiten selbstständig. Es ist daher sehr kompliziert diese gegen Testkomponenten zu ersetzen.

Im NuPTools-Projekt wird trotz der Gefahren ein Service Locator eingesetzt. Da mehrere Plug-ins gleichzeitig zum Einsatz kommen können, existieren auch mehrere Einstiegspunkte. Der Container muss daher von verschiedenen Codestellen erreichbar sein. Um die genannten Gefahren zu vermeiden, darf der Service Locator jedoch nicht freizügig von jeder Komponente verwendet werden. Die Nutzung ist auf die Einstiegspunkte der einzelnen Plug-ins beschränkt.

5.10 Grafische Benutzerschnittstellen

Bisher wurden grafische Benutzerschnittstellen im NuPTools-Projekt mit dem Windows-Forms-Frameworks erstellt. Zukünftig werden diese mit dem Windows Presentation Foundation (WPF)-Framework erstellt. Dieses ermöglicht als Nachfolger des Windows-Forms-Frameworks modernere und funktionalere Oberflächen. WPF wird üblicherweise mit dem Model-View-ViewModel-Muster(MVVM) angewandt. Eine Beschreibung des Musters gibt Microsoft (2012).

Das Muster verwendet die drei in Abbildung 5.12 zu sehenden Objekttypen. Das View-Objekt dient der Definition der Oberfläche. Es darf zwar Darstellungslogik enthalten, jedoch keine Fachlogik oder Daten. Beispielsweise könnte ein Datum anstelle eines formatierter Text als Datumsobjekt an die Oberfläche übergeben werden. Das für die Darstellung gewünschte Format kann in dem Fall durch die View festgelegt werden.

Daten und Fachlogik sind vorwiegend in dem bzw. den Model-Objekten enthalten. Diese stellen die funktionale Seite des Musters dar. Verbunden werden die Objekte durch das ViewModel. Dieses bereitet die Daten der Models für die View auf und steuert die Fachlogik.

Die Kommunikation zwischen der View und dem ViewModel erfolgt nach dem Beobachtermuster. Dieses kommt in der Publish/Subscribe-Variante zum Einsatz. Hierbei wird die View mittels Ereignissen über geänderte Daten informiert und kann auf die Änderungen mit einer Aktualisierung der Oberfläche reagieren. Eine detailliertere Beschreibung des Muster folgt in Abschnitt 5.11 auf der nächsten Seite. Auch von den Models zu dem

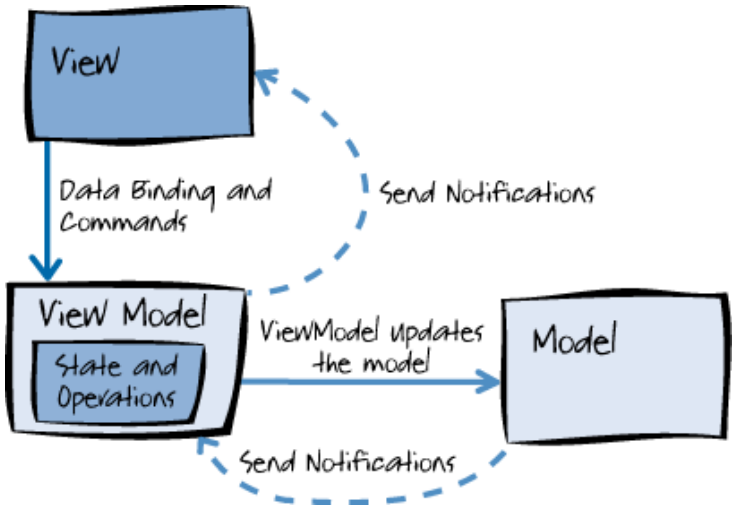

Abbildung 5.12: Objekte des MVVM-Muster (Microsoft 2012)

ViewModel kann eine solche Kommunikation eingerichtet werden. Als steuernde Komponente des Musters ist das ViewModel in der Regel der Auslöser für Datenänderungen. Das Beobachtermuster muss daher nur selten in den Models implementiert werden.

Der Vorteil des MVVM-Musters ist die sehr lose Kopplung zwischen dem View und dem ViewModel bzw. die vollständige Entkopplung des Views von den Models. Durch diese starke Trennung können die Oberfläche und die Fachlogik getrennt voneinander entwickelt werden. Beispielsweise kann eine neue Oberfläche für die selben Daten ohne Änderungen an den Models oder dem ViewModel entwickelt werden. Ein anderes Beispiel ist die Aufteilung der Entwicklung auf verschiedene Mitarbeitern wie Oberflächendesigner und Softwareentwickler.

Weiterer Vorteil der starken Trennung ist die Testbarkeit. Durch die verringerten Abhängigkeiten können Komponententests einfacher durchgeführt werden. Zusätzlich fungiert das ViewModel als Adapter zu den Models. Änderungen an den Models können somit ohne Änderung an der View durchgeführt werden.

5.11 Messenger

Die Messenger-Komponente ist Teil des DevExpress MVVM Frameworks[9], das im NuPTools-Projekt verwendet wird. Es implementiert ein Beobachter-Muster, wie es von Geirhos (2015) beschrieben wird. Die Verwendung der Komponente wird von Alexander (DevExpress Support) (2013) beschrieben.

Bei dem Beobachter-Muster handelt es sich um ein Kommunikationsmuster, mit dessen Hilfe Komponenten unabhängig voneinander kommunizieren können. Das Muster kann über vier unterschiedliche Wege Nachrichten senden.

1. Push: Die Senderkomponente sendet die Nachricht an die ihr bekannten Empfänger.
2. Pull: Die Empfängerkomponente ruft Nachrichten von dem Sender ab. Dies geschieht in der Regel in Intervallen.
3. Publish/Subscribe: Die Empfängerkomponente registriert sich bei dem Sender. Dieser sendet seine Benachrichtigungen an alle registrierten Empfänger.

[9] https://www.devexpress.com/Products/NET/Controls/WPF/

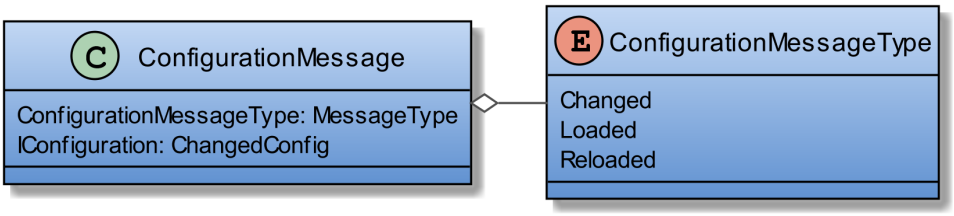

Abbildung 5.13: Beispiel eines Nachrichtentyps

4. Message Bus: Der Empfänger registriert sich an einem Nachrichtenbus. Sender versenden ihre Nachrichten über diesen Nachrichtenbus. Die Verteilung der Nachrichten an die entsprechenden Empfänger obliegt dem Nachrichtenbus.

Der Messenger des DevExpress Frameworks implementiert das Muster in Form eines Nachrichtenbus. Als Nachricht kann prinzipiell jedes Objekt versendet werden. Es ist jedoch empfehlenswert einen eigenen Nachrichtentyp zu implementieren. Dies erlaubt eine genauere Beschreibung der Nachricht. Beispielsweise kann beim Versenden eines Konfigurationsobjektes nicht interpretiert werden, ob die Konfiguration erstmalig geladen, erneut geladen oder geändert wurde. Ein Datentransferobjekt, das die Daten des Objektes für die Übermittlung umformt oder vereinfacht, ist jedoch nicht nötig. Das Nachrichtenobjekt kann, wie in Abbildung 5.13 zu sehen, bei Bedarf neben einer Beschreibung ebenfalls die betreffenden Daten und Objekte beinhalten.

Zum Empfangen muss sich der Empfänger bei dem Messenger für einen Nachrichtentyp registrieren. Die dabei angegebene Ereignismethode wird beim Senden von Nachrichten von dem Messenger aufgerufen. Für das Versenden von Nachrichten muss der Sender nur ein Objekt des entsprechenden Nachrichtentyps erstellen und dem Messenger übergeben.

5.12 Konfigurationsmanager

Der Konfigurationsmanager dient der Verwaltung der Plug-in-Konfigurationen. Er löst die bisher für jedes Plug-in separat entwickelte Konfigurationsverwaltung ab. Zusätzlich bietet er eine Oberfläche zur Verwaltung der Plug-in-Konfigurationen durch den Anwender.

Die historischen Konfigurationsverwaltungen ließen jeden Spielraum hinsichtlich der Konfigurationsart. Hieraus resultierten verschiedene Speicherformate und -orte. Zu den Aufgaben des Konfigurationsmanagers gehören das Speichern und Laden der Konfigurationsdateien von definierten Dateipfaden. Durch die zentrale Verwaltung wird die Varianz in

den Speicherorten deutlich reduziert und alle Einstellungen in einem einheitlichen Format gespeichert.

Die Konfigurationen der verschiedenen Plug-ins können dabei getrennt oder in einer gemeinsamen Datei gespeichert werden. Auch die gemeinsame Verwendung einer Konfigurationsdatei durch mehrere Anwender wird durch den Manager ermöglicht. Hierzu kann eine Weiterleitung zu einer Konfigurationsdatei auf einem Netzlaufwerk eingerichtet werden.

Aus Anwendersicht ist die Konfiguration der Plug-ins deutlich vereinfacht. Bei dem bestehenden Verfahren müssen Konfigurationsänderungen von dem Anwender oder einem Dienstleister händisch in der Konfigurationsdatei vorgenommen werden. Um diesen fehleranfällige Vorgang zu vermeiden stellt der Konfigurationsmanager eine Konfigurationsoberfläche bereit. Plug-ins müssen hierfür nur eine Maske für ihre Konfigurationsobjekte bereitstellen.

Auch aus Entwicklersicht ist die Verwendung des Konfigurationsmanagers vereinfacht worden. Im bestehenden System muss der Entwickler die Datei händisch auslesen. Dies geschieht in de Regel initial zum Plug-in-Start. Häufig wird dies aber auch mehrfach in beliebigen Prozessschritten durchgeführt. Durch den Konfigurationsmanager muss nur der entsprechende Konfigurationstyp von dem Manager angefordert werden. Dieser übernimmt sämtliche Verwaltungsaufgaben und stellt ein geladenes oder neues Konfigurationsobjekt zur Verfügung.

5.13 Hilfsmethoden

Das NuPFramework beinhaltet einen großen Umfang an Hilfsmethoden. Diese werden mit Erweiterungsmethoden (Extensions) umgesetzt. Die C#-Erweiterungsmethoden sind eine Umsetzung der von Bracha und Cook (1990) beschriebenen Mixins. Bei Mixins handelt es sich um Methodendefinitionen, die einen Typen erweitern ohne von Vererbungsmechanismen Gebrauch zu machen.

Die Anwendung von Erweiterungsmethoden beschreibt Hall (2015). Extensions werden in einer statischen Klasse als statische Methode implementiert. Der erste Methodenparameter erhält das zusätzliche Schlüsselwort *this*. Durch dieses wird die Methode als Extension markiert. Gleichzeitig wird durch das Schlüsselwort der zu erweiternde Typ definiert.

Das Beispiel in Ausschnitt 5.2 auf der nächsten Seite zeigt eine Mittelwert-Erweiterung für alle Auflistungen von *double*-Aufzählungen. Der Klassenname kann frei gewählt werden.

```
public static class IEnumerableExtensions
{
    public static double Median(this IEnumerable<double> xs)
    {
        var ys = xs.OrderBy(x => x).ToList();
        var mid = (ys.Count - 1) / 2.0;
        return (ys[(int)(mid)] + ys[(int)(mid + 0.5)]) / 2;
    }
}
```

Ausschnitt 5.2: Eine Erweiterungsmethode zur Erweiterung von Listen aus *double*-Werten zur Berechnung des Mittelwertes

Häufig werden die Methoden jedoch, wie im Beispiel auch, nach dem zu erweiternden Typ gruppiert und in entsprechenden Klassen mit der Suffix „Extensions“ angeordnet. Durch den mit dem Schlüsselwort markierten Parameter kann die Methode an allen Typen angewandt werden, die *IEnumerable* mit dem generischen Typparameter *double* implementieren.

Kapitel 6

Zusammenfassung

Im Rahmen der Arbeit wurde das NuPTools-Projekt modernisiert. Dabei wurden sowohl die Arbeitsweisen als auch die Software analysiert und zu verschiedenen Frameworks neu organisiert. Erstmalig wurde eine automatisierte Qualitätssicherung des Projektes umgesetzt. Durch die Modernisierung wurden die bisher versteckten technischen Schulden aufgedeckt und Werkzeuge zu deren Bekämpfung/Vermeidung eingeführt.

Als Basis des Erstellungsprozess dient ein TeamCity-System. Mit diesem wird ein Continuous Delivery-Verfahren realisiert. Das Verfahren übernimmt die Erstellung aller Komponenten. Dies nimmt den Entwicklern einige zeitaufwendige Routineaufgaben und befreit dadurch bereits erste Entwicklungszeiten. Gleichzeitig dient das TeamCity-System auch als Verwaltung für die Softwareversionen. Der „Verlust“ von ausgelieferten Produkten wird dadurch vermieden. Durch diese Archivierung alter Produktversionen stehen diese bei Bedarf allen Entwicklern zur Fehlersuche zur Verfügung.

Ein Teil des Continuous Delivery-Prozesses sind automatisierte Softwaretests. Die Test werden von dem Server autonom ausgeführt. Dies ermöglicht die regelmäßige Überprüfung der Funktionalität. So wird die Anzahl an manuellen Tests reduziert und weitere Entwicklungszeit befreit. Durch die Tests wird die Prüfung von einzelnen Komponenten ermöglicht. Auch die Lauffähigkeit der Plug-ins wird automatisiert getestet. Hierfür werden Installationstests durchgeführt und so die Lauffähigkeit einzelner Plug-ins und Kombinationen mehrerer installierter Plug-ins sichergestellt.

Die Codequalität wird besonders durch die Codeinspektionen erhöht. Hierbei werden verschiedene Qualitätsmängel aufgedeckt. Dies umfasst Widersprüche gegen einen verbreiteten Codestil und programmiersprachliche Verbesserungsmöglichkeiten. Ebenfalls werden Codedubletten im gesamten Projekt gesucht. Durch ausgewählte Metriken wie die kogni-

tive Komplexität werden Messwerte für die Codequalität ermittelt. Die Inspektion stellt somit ein wichtiges Kontrollwerkzeug zur Reduzierung der technischen Schulden dar.

Zur Verkürzung der Aktualisierungszyklen werden die NuPTools um eine Installations- und Aktualisierungskomponente erweitert. Durch diese wird der Bedarf an Dienstleistern bei einer Produktaktualisierung minimiert. Dies zieht auch die Möglichkeit mit sich, mit den NuPTools einen größeren Markt zu bedienen. Da für die Installation keine Dienstleister benötigt werden können mehr Kunden bedient werden. Außerdem werden bei fremdsprachigen Kunden keine entsprechend sprachlich befähigten Dienstleister benötigt. Die Installations- und Aktualisierungskomponente ist somit eine wertvolle Grundlage für die Erweiterung auf den internationalen Markt.

Parallel zu den genannten Modernisierungsschritten werden die Architektur und diverse Technologien erneuert. Die starre Altarchitektur wird gegen eine lose gekoppelte Architektur ausgetauscht. Hierdurch wird die Wartbarkeit stark verbessert. Ebenso werden einige Komponenten, wie beispielsweise die Oberflächen, durch modernere Technologien ausgetauscht.

Die Modernisierungsschritte transformieren das NuPTools-Projekt von einem historisch verwachsenen, kaum pflegbarem zu einem modernen und flexiblen Projekt. Für die Entwicklung neuer und innovativer Plug-ins werden freie Zeiten und technische Möglichkeiten geschaffen.

Das Projekt bietet besonders in technologischen Aspekten weitere Modernisierungsmöglichkeiten. Diese Verbesserungen sind für die folgende Entwicklung vorgesehen. Ebenso wie die Umsetzung der Modernisierungen in Teilprojekten, die im Zeitraum dieser Arbeit keinen Anpassungsbedarf hatten. Durch neue Versionen des TeamCity-Servers kann der Continuous Delivery-Prozess vereinfacht und optimiert werden. Auch dies ist für die Zukunft vorgesehen.

Anhang A

Datenträger

Der beigefügte Datenträger enthält die folgenden Daten:

- Eine digitale Kopie dieser Arbeit
- Den im Rahmen der Arbeit entwickelten Generator für PlantUML-Diagramme
- Ein Beispielplug-in, das den im Rahmen der Arbeit ausgearbeiteten Richtlinien entspricht
- Inhaltliche Kopien aller verwendeten Internet-Quellen

Literatur

.NET Foundation (2017). *NuGet Gallery | Home*. URL: https://www.nuget.org/ (besucht am 25. 08. 2017).

Alexander (DevExpress Support) (13. Dez. 2013). *DevExpress MVVM Framework. Interaction of ViewModels. Messenger. - WPF*. URL: https://community.devexpress.com/blogs/wpf/archive/2013/12/13/devexpress-mvvm-framework-interaction-of-viewmodels-messenger.aspx (besucht am 11. 10. 2017).

Bracha, Gilad und William Cook (1990). „Mixin-based inheritance“. In: *Proc. OOPSLA '90*. ACM Press, S. 303–311.

Campbell, G. Ann (März 2017). *Cognitive Complexity - A new way of measuring understandability*. Techn. Ber. SonarSource S.A.

Cardella, Carlo (22. Feb. 2008). *Version numbers in a compiled assembly*. URL: https://blogs.msdn.microsoft.com/carloc/2008/02/22/version-numbers-in-a-compiled-assembly/.

Cunningham, Ward (22. Jan. 2011). *Ward Explains Debt Metaphor*. URL: http://wiki.c2.com/?WardExplainsDebtMetaphor (besucht am 05. 09. 2017).

Cwalina, Krzysztof und Brad Abrams (2008). *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (Microsoft Windows Development Series)*. Addison-Wesley Professional. ISBN: 0-321-54561-3. URL: https://www.amazon.com/Framework-Design-Guidelines-Conventions-Development-ebook/dp/B0017SWPNO?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B0017SWPNO.

Driessen, Vincent (5. Jan. 2015). *A successful Git branching model*. URL: http://nvie.com/posts/a-successful-git-branching-model/ (besucht am 25. 08. 2017).

Duvall, Paul M., Steve Matyas und Andrew Glover (2009). *Continuous integration : Improving software quality and reducing risk*. 4. print. Includes bibliographical references and index. Upper Saddle River, NJ: Addison-Wesley, XXXII, 283 S. ISBN: 9780321336385. URL: https://www.katalog.fh-zwickau.de/Record/0000854371.

FakeItEasy (2017). *FakeItEasy - It's faking amazing*. URL: https://fakeiteasy.github.io/ (besucht am 12. 10. 2017).

Fowler, Martin (30. März 2013). *Continuous Delivery*. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (besucht am 04. 05. 2017).

Geirhos, Matthias (11. Juni 2015). *Entwurfsmuster*. Rheinwerk Verlag GmbH. 643 S. ISBN: 3836227622. URL: http://www.ebook.de/de/product/22387366/matthias_geirhos_entwurfsmuster.html.

Gregsdennis (21. Jan. 2016). *How to Version Assemblies Destined for Nuget*. URL: https://codingforsmarties.wordpress.com/2016/01/21/how-to-version-assemblies-destined-for-nuget/ (besucht am 08. 06. 2017).

Hall, Garry McLean (11. Mai 2015). *Agile Softwareentwicklung mit C Sharp*. Dpunkt.Verlag GmbH. ISBN: 3864902851. URL: http://www.ebook.de/de/product/23907498/garry_mclean_hall_agile_softwareentwicklung_mit_c_sharp.html.

Mauer, George (1. Mai 2015). *Why Not MsTest*. URL: http://georgemauer.net/2015/05/01/why-not-mstest (besucht am 14. 06. 2017).

Megorskaya, Irina (18. Nov. 2016). *Build Agent*. URL: https://confluence.jetbrains.com/display/TCD10/Build+Agent (besucht am 25. 08. 2017).

Microsoft (10. Feb. 2012). *The MVVM Pattern*. URL: https://msdn.microsoft.com/en-us/library/hh848246.aspx (besucht am 12. 10. 2017).

Microsoft Corporation (30. März 2017). *Names of Namespaces | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces (besucht am 02. 11. 2017).

NUnit Software (2017). *NUnit Documentation*. URL: https://github.com/nunit/docs/wiki/.

Osherove, Roy (5. Dez. 2013). *The Art of Unit Testing*. Manning. ISBN: 1617290890. URL: http://www.ebook.de/de/product/20470824/roy_osherove_the_art_of_unit_testing.html.

Palme, Daniel (2. Sep. 2017). *IoC Container Benchmark - Performance comparison*. URL: http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison (besucht am 04. 10. 2017).

Pohl, Klaus und Chris Rupp (2015). *Basiswissen Requirements Engineering : Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering"; Foundation Level nach IREB-Standard*. 4., überarb. Aufl. Literaturangaben. Heidelberg: dpunkt-Verl., XIX, 171 S. ISBN: 9783864902833. URL: http://d-nb.info/1069136069/04.

Rupp, Chris und Stefan Queins (11. Apr. 2012). *UML 2 glasklar*. Hanser Fachbuchverlag. ISBN: 3446430571. URL: http://www.ebook.de/de/product/18112464/chris_rupp_stefan_queins_uml_2_glasklar.html.

Sato, Danilo (25. Juni 2014). *CanaryRelease*. URL: https://martinfowler.com/bliki/CanaryRelease.html (besucht am 15. 05. 2017).

Seemann, Mark (26. Apr. 2010). *Why I'm migrating from MSTest to xUnit.net*. URL: http://blog.ploeh.dk/2010/04/26/WhyImmigratingfromMSTesttoxUnit.net/ (besucht am 14. 06. 2017).

Simple Injector Contributors (2017). *Advanced Scenarios — Simple Injector 4 documentation*. URL: http://simpleinjector.readthedocs.io/en/latest/advanced.html#property-injection (besucht am 04. 10. 2017).

Spillner, Andreas und Tilo Linz (11. Sep. 2012). *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH. ISBN: 3864900247. URL: http://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html.

Starke, Gernot (7. Juli 2015). *Effektive Softwarearchitekturen*. Hanser Fachbuchverlag. ISBN: 3446443614. URL: http://www.ebook.de/de/product/24327951/gernot_starke_effektive_softwarearchitekturen.html.

Wikipedia (1. Dez. 2016a). *Autodesk Vault — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Autodesk_Vault&oldid=752439019 (besucht am 16. 05. 2017).

— (23. Nov. 2016b). *Domänenspezifische Sprache — Wikipedia, Die freie Enzyklopädie*. URL: https://de.wikipedia.org/w/index.php?title=Dom%C3%A4nenspezifische_Sprache&oldid=159987491 (besucht am 25. 09. 2017).

— (18. Apr. 2017a). *Autodesk Inventor — Wikipedia, Die freie Enzyklopädie*. URL: https://de.wikipedia.org/w/index.php?title=Autodesk_Inventor&oldid=164688556 (besucht am 16. 05. 2017).

— (17. März 2017b). *Common Intermediate Language — Wikipedia, Die freie Enzyklopädie*. URL: https://de.wikipedia.org/w/index.php?title=Common_Intermediate_Language&oldid=163683509 (besucht am 05. 09. 2017).

Wolff, Eberhard (2014). *Continuous Delivery: Der pragmatische Einstieg (German Edition)*. dpunkt.verlag. ISBN: 978-3-86490-208-6.

Erklärung der eigenständigen Verfassung der Masterarbeit

Ich versichere, dass ich die Masterarbeit selbstständig verfasst und keine, als die von mir angegebenen Hilfsmittel benutzt und bei Zitaten die Quellen kenntlich gemacht habe.

Zwickau, 03. Januar 2018

Andreas Brandhoff