ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Ausarbeitung für das Pflichtmodul Software-Entwicklungsprozesse
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Software-Modernisierung

von

Nicolas Klein, Alexander Stolz

begutachtet von

Prof. Dr. Helmut Folz

Saarbrücken, 29. August 2021

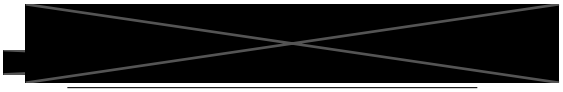
Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 29. August 2021



Nicolas Klein, Alexander Stolz

Inhaltsverzeichnis

1 Einleitung

Sofern eine Software in Benutzung ist, muss sie angepasst und verbessert werden. [21]
Dies zeigte sich bei der Einführung der DSGVO [8]. Letztere stellte unter anderem neue Anforderung an den Umgang mit personenbezogenen Daten. Bestehende Software-Systeme müssen sich diesen Anforderungen stellen. Wegen nicht-Beachtung dieser Anforderungen musste der Bekleidungshersteller H&M im Jahr 2020 ein Bußgeld von 35,3 Millionen Euro zahlen [29].
Um auf äußere Einflüsse zu reagieren, muss eine Software folglich stets flexibel bleiben. Eine flexible Software kann auch an neue Anforderungen angepasst werden. Allerdings widerstehen, über viele Jahre gewachsene Software-Systeme, meist Änderungen.

Die Einführung der DSGVO zeigt anschaulich, dass diese Flexibilität einer Software essenziell für den Betrieb und Verkauf dieser ist. Neue Anforderungen können durch den Gesetzgeber formuliert werden. Anschließend muss diesen Folge geleistet werden.
Neben dem Gesetzgeber muss die Software auch flexibel auf die Konkurrenz reagieren können. Neue Vertriebskonzepte wie Software as a Service (SaaS) halten zunehmend mehr Einzug in der IT-Branche. Um diesen Markt bedienen zu können, muss die Software ebenfalls angepasst werden.
Die Frage nach der Flexibilität einer Software ist folglich nicht nebensächlich. Eine unflexible Software wird zunehmend weniger wettbewerbsfähig. Um am Markt relevant zu bleiben, muss die Flexibilität einer Software erhalten bleiben.

In dieser Arbeit werden die Gründe für eine Software-Modernisierung aufgearbeitet und konkrete Maßnahmen für eine Modernisierung der Software betrachten.

1.1 Struktur der Arbeit

Die Arbeit ist dabei wie folgt strukturiert. Zunächst wird in Kapitel 2 analysiert, warum eine Software-Modernisierung benötigt wird. Kapitel 3 ordnet die Software-Modernisierung in den Software-Entwicklungszyklus ein. In Kapitel 4 wird auf das Refactoring und dessen Bedeutung für die Software-Modernisierung eingegangen. Kapitel 5 wird das Cloud-Computing betrachten und dessen Bedeutung für die Software-Modernisierung aufzeigen. Im letzten Kapitel wird abschließend ein konkretes Vorgehensmodell für eine Modernisierung mittels des Cloud-Computings vorgestellt.

2 Lehmanns-Gesetze

In diesem Kapitel wird auf Lehmanns Gesetze eingegangen. Hierbei werden zunächst die Personen Lehmann und Bélády betrachtet.
Anschließend werden das erste und zweite der Gesetze von Lehmann und Bélády erläutert. Mit diesem Wissen können einige inneren und äußeren Ursachen für Lehmanns Gesetze analysiert werden.
Das Kapitel schließt mit den langfristigen Auswirkungen von Lehmanns Gesetzen.

2.1 Die Personen Lehmann und Bélády

Die Zusammenhänge zwischen Benutzung und Anpassung der Software wurden schon früh in der Informatik erkannt. Der Informatik-Professor Manny Lehman und der Informatiker László Bélády formulierten zwischen 1974 und 1996 eine Reihe von Gesetzen. Diese nannten sie die "Laws of Software-Evolution". Die Gesetze beschreiben ein Gleichgewicht zwischen Kräften, die neue Entwicklungen vorantreiben, auf der einen Seite und Kräften, die den Fortschritt bremsen, auf der anderen Seite. [21]

2.1.1 Korrektheit von Lehmanns Gesetzen

Lehmanns und Bélády's Gesetze sind weniger Gesetze, als es Beobachtungen sind. Die Korrektheit dieser Beobachtungen wurde bereits in vielen Systemen und Anwendungen untersucht und validiert.
Ein Beispiel dafür wäre die Arbeit von Feitelson et al. [15]. In dieser wurden 810 Version des Linux-Kernels, welche über ein Zeitraum von 14 Jahren veröffentlicht wurden, nach der Evolution des Software-Systems charakterisiert. Als Basis wurden Lehmanns Gesetze angenommen und diese mit verschiedensten Metriken nachgewiesen.
Zum Beispiel wurde das System-Wachstum anhand von Codezeilen oder der Anzahl von Funktionen quantifiziert. Darüber hinaus wurde das funktionale Wachstum von Linux anhand der Anzahl von Systemaufrufen gemessen.
Dabei konnten das erste und zweite von Lehmanns Gesetzen nachgewiesen werden. Auf diese wird im Folgenden eingegangen.

2.1.2 Das erste und zweite Gesetz

Lehmann und Bélády haben im ersten Gesetz schon 1974 erkannt, dass eine Änderung und Umstrukturierung eines Software-Systems stattfindet. Vorausgesetzt das Software-System wird aktiv genutzt. Ferner wurde im zweiten Gesetz erkannt, dass der Effekt dieser Änderung negativ ist. Außer es wird ein zusätzlicher Aufwand unternommen, um dem entgegenzuwirken.

2.2 Gründe für Lehmanns Gesetze

Nachdem Feitelson et al. die Gültigkeit von Lehmanns Gesetzen festgestellt hat, gilt es zu fragen, warum diese Gesetze gelten. Ausgangspunkt dafür sind äußere und innere Kräfte, welche auf die Software einwirken.
Auf diese beiden Kräfte wird im Folgenden eingegangen.

2.2.1 Äußere Gründe für Lehmanns Gesetze

Ein Softwaresystem ist nicht isoliert zu betrachten. Es ist in eine Umgebung eingebettet. Grundsätzlich besteht ein Bedarf an Anpassungen, wenn sich die Umgebung der Software wandelt.[4] Hierfür werden im Folgenden einige mögliche Gründe aufgeführt:

- Beispiel für die fachliche Umgebung einer Software wären neue Compliance-Anforderungen. Wird eine Software über viele Jahre oder Jahrzehnte genutzt, sind Gesetzesänderungen zu erwarten. Folglich muss eine, sich in der Benutzung befindende, Software darauf reagieren.
- Auch ein Umdenken in der Geschäftsstrategie ist ein möglicher äußerer Faktor. Neue Märkte haben neue Anforderungen. Ein Beispiel hierfür wäre das Expandieren nach China. Im fern-östlichen Raum gelten nicht nur andere Gesetze, sondern auch andere Sitten. Dies kann umfangreiche Änderungen an der Software nach sich ziehen.
- Darüber hinaus kann auch ein neues Vertriebsmodell für Veränderung sorgen. Beispiele hierfür wären Geschäftsmodelle wie Software as a Service (SaaS). Auf SaaS wird im Rahmen des letzten Kapitels umfangreicher eingegangen.
- Abschließend kann auch ein Fortschritt der Technik Auslöser für eine Reihe von neuen Anforderungen sein. Neue Programmiersprachen und Frameworks können ein schnelleres Voranschreiten in der Entwicklung ermöglichen. Letzteres wiederum kann so die Zeit zwischen der Entwicklung und dem ersten Release verkürzen. Auch Fehler im System können möglicherweise schneller behoben werden. Sollte die Konkurrenz die genannten Vorteile erzielen, kann dies ein Wettbewerbsvorteil für die Konkurrenz sein.

2.2.2 Innere Gründe für Lehmanns Gesetze

Neben den bereits beschriebenen äußeren Einflüssen führen auch eine Reihe von inneren Gründen zu Lehmanns Gesetzen. Die Software erfährt auch ein Wandel der technischen Umgebung. Im Folgenden werden hierfür einige Beispiele angeführt:

- Seacord et al. [26] führt als ersten Grund Verbesserungen an dem Produkt selbst an. Hierzu gehört Fehlerverbesserungen und Performance-Optimierungen. Neue Komponenten werden hinzugefügt, existierende Komponenten angepasst. [26]
- Während der Definitions-Phase der Architektur kann ein Grad von Anpassungsfähigkeit und Flexibilität eingeplant werden. Jedoch sind nicht die Anforderungen bekannt, welche erst in der Zukunft gestellt werden. Die Architektur kann nicht für diese ausgelegt werden. Folglich degeneriert die Architektur zunehmend.[26][7]
- Darüber hinaus wächst die Code-Basis. Wiederholte Modifikation führt meist zu mehr Code und komplexerer Software. Das System wird folglich brüchiger. Die

Wahrscheinlichkeit von Seiteneffekten nimmt zu, mit jeder Änderung die eingearbeitet wird. Die Wartung der Software wird folglich Schritt für Schritt schwieriger. Letzteres verlangt nach mehr Personal und nach speziell geschulten Personal. [26][7]

- Abschließend ist es möglich, dass neue Mitarbeiter an der Software arbeiten. Die ursprünglichen Mitarbeiter, welche die Architektur der Software entwickelt haben, sind möglicherweise nicht mehr in der Firma beschäftigt oder im Ruhestand. Spätestens jetzt können Verständnisprobleme mit der Definition der Architektur oder der bestehenden Code-Basis aufkommen.
  Fehlende Dokumentation, gealterte Programmiersprachen oder das Missachten von Programmierrichtlinien in der Vergangenheit verstärken diesen Effekt.
  Seit 1970 sind die Kosten für die Wartung so weit gestiegen, dass sie die Kosten der initialen Entwicklung übersteigen. Grafik 2.1 verdeutlicht dies.

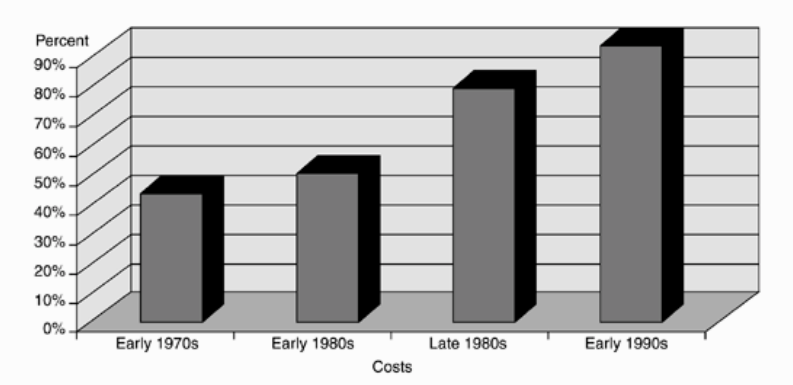


Abbildung 2.1: Software-Kosten, welche der Software-Modernisierung gewidmet werden. [26]

2.3 Langfristige Auswirkungen von Lehmanns Gesetzen

Wird das System nicht an die gewandelte Umgebung angepasst, entsteht zunehmen eine Diskrepanz zwischen der Umgebung der Software und dem Softwaresystem selbst. Grafik 2.2 illustriert diesen Zusammenhang.
Fehlt den Mitarbeitern das Verständnis für die Architektur und bestehende Codebasis, wird es zunehmend schwieriger die Software anzupassen. Möglicherweise ist die Software nicht umfassend testbar. Änderungen können folglich unvorhergesehene Wechselwirkungen hervorrufen. [7]
Dies führt unter den Mitarbeitern zu dem sogenannten "Fear-Driven-Development"[7]. Anpassungen werden dadurch nur spärlich eingearbeitet und meist vermieden.
Die Software wird demnach resistent gegen Änderungen am Code. Eine Erhaltung oder Steigerung der Effizienz setzt allerdings Änderungen voraus. Darüber hinaus sind Fehlerursachen ohne umfassende Kenntnisse der Codebasis schwieriger auszumachen. Die Zeit und Kosten Fehler zu beseitigen steigen.
Zusammenfassend lässt sich also sagen, dass die Software unwirtschaftlicher wird. Immer größere Investitionen werden nötig um immer weniger Steigerungen in der Nützlichkeit der Software zu erreichen. Folglich wird die Software von Konkurrenz-Produkten überholt und zunehmend weniger konkurrenzfähig.

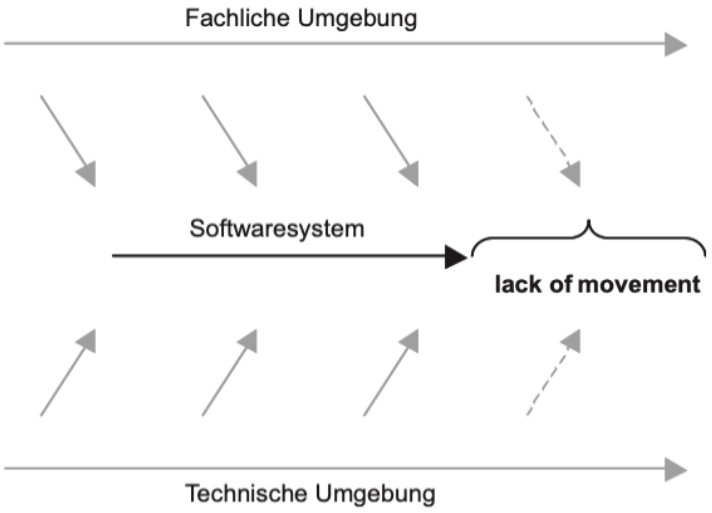


Abbildung 2.2: Folgen einer fehlenden Anpassung der Software an ihre Umgebung [4]

2.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass sich die Problemstellung der Software-Alterung auf zwei verschiedene Aspekte bezieht.
Zunächst wird durch Lehmanns Gesetze klar, dass eine Software angepasst werden muss. Sonst schreitet das Umfeld der Anwendung fort, während die Anwendung stehen bleibt. Die Software veraltet. Führt man allerdings Änderungen an der Software durch Ergeben sich neue Probleme. Denn werden die Grundprinzipien der Architektur missachtet, degeneriert die Architektur. Diese Degeneration führt erneut zu einer Alterung der Software. Folglich stellt sich die Frage, wie dieser Alterung entgegengewirkt werden kann, ohne die Architektur zu schädigen.

3 Die Software-Entwicklung

Das vorherige Kapitel führte Lehmanns Gesetze ein. Basierend auf deren Annahmen wurden innere und äußere Gründe herausgearbeitet, welche zu diesen Gesetzen führen. Abschließend wurden einige Auswirkungen dieser Gesetze aufgezeigt.
In diesem Kapitel wird dargestellt, was Software Modernisierung ist. Dafür wird zunächst der Softwareentwicklungsprozess nach Seacord et al. dargestellt. Anschließend wird die Software-Modernisierung in Seacord's Entwicklungsprozess eingeordnet.

3.1 Der Software-Entwicklungsprozess

Die Software-Modernisierung kann als Teil des Software-Entwicklungsprozesses betrachtet werden. Seacord et al. [26] teilt diese in 3 voneinander getrennte Phasen auf.
Hierzu zählt zunächst die Instandhaltung der Software. Diese geht über in die Software-Modernisierung. Bis schließlich eine Ersetzung des Systems angestrebt wird. Grafik 3.1 illustriert diese 3 Phasen.
Die gepunktete Linie in Grafik 3.1 zeigt die Anforderungen der Firma, die sogenannten *Business needs*. Die durchgezogene Linie hingegen gibt die Funktionalität des Systems an. Eine wiederholte Instandhaltung der Software nährt die beiden Linien kurzfristig an. Die Funktionalität ist in diesen Zeitspannen ausreichend um die Business Needs zu befriedigen. Mit der Alterung des Systems können zunehmend solche kurzfristigen Lösungen nicht mehr die Funktionalität ausreichend steigern.
Eine Modernisierung der Software nimmt mehr Zeit und Geld in Anspruch, als eine Wartung. Mit umfangreichen Änderungen des Systems kann allerdings erneut die Funktionalität an die Anforderungen angeglichen werden. Wenn das alte System schließlich nicht mehr weiterentwickelt werden kann, muss es ersetzt werden. Die Entscheidung, welche der drei Optionen für ein bestimmtes System angewandt wird, kann nur durch eine eingehende Analyse bestimmt werden. Zeit, Kosten und Abhängigkeiten zur Software sind nur ein kleiner Teil der relevanten Parameter. Oft sind Systemausfälle im Zusammenhang mit längeren Modernisierungsmaßnahmen schlicht nicht akzeptabel. [26]
Im Folgenden werden die Unterschiede von Wartung, Modernisierung und Ersetzung aufgezeigt.

3.2 Software-Wartung

Die Wartung arbeitet kleine Verbesserungen, wie Fehlerbehebungen oder strukturelle Verbesserungen ein. Diese Anpassungen verändern nichts an der grundsätzlichen Struktur der Software. Folglich bleibt die eigentliche Architektur unangetastet. [26][4]

Der Prozess der Wartung ist iterativ. Sie unterstützt die Entwicklung des Systems, hat aber auch Einschränkungen, denn sie erschließt keine Wettbewerbsvorteile. [26] Der Umzug in die Cloud oder die Umstellung des Softwareangebots auf Software as a Service sind keine Wartungs-Operationen. Letztere erfordern tiefgreifende strukturelle Anpassungen und können die Architektur des Software-Systems beeinflussen.

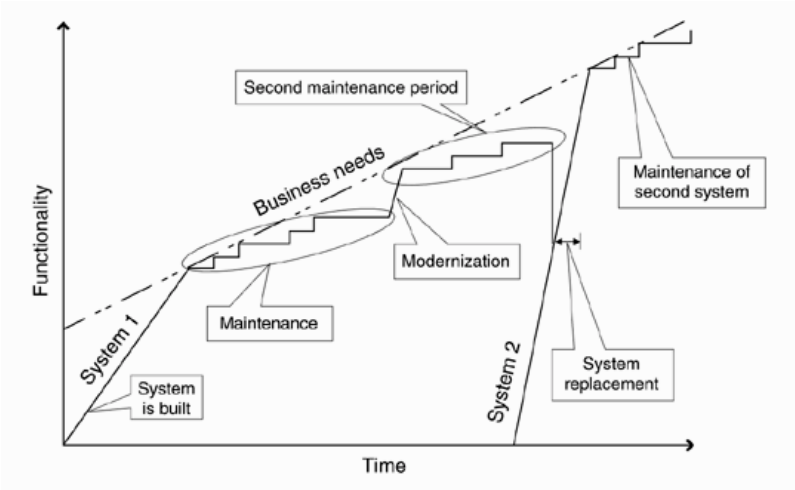


Abbildung 3.1: Lebenszyklus eines Informationssystems [26]

Allerdings lässt sich die Software-Wartung in Kategorien aufteilen. Diese sind in der Norm IEEE610 festgehalten.:

korrektive Wartung
Boomer et Al. beschreibt diese, als das Beheben von Softwarefehlern. Letztere sind Restfehler, welche nicht während der Entwicklung bemerkt wurden. Die Art des Fehlers kann weiter als Mängel oder Instandhaltung verstanden werden. Ersteres sind kleine Fehler, welche gesammelt und zu gegebener Zeit behoben werden können. Instandhaltung meint allerdings schwere Mängel, welche den Betrieb stören können. [4]

adaptiver Wartung
Bei der adaptiven Wartung werden kleine fachliche oder technischen Anpassungen unternommen. Für erstere wäre die Änderung des Mehrwertsteuersatzes zu nennen. Es kann auf eine neue System-Version umgestellt werden. Diese System-Umstellung ist lediglich dazu da, denn vorherigen Zustand des Systems wiederherzustellen. Hierbei entsteht kein Mehrwert. [4]

perfektionierender Wartung
Das Ziel der perfektionierenden Wartung ist die Verbesserung nicht-funktionaler Anforderungen. Mit kleinen Änderungen wird dabei z.B. die Antwortzeit der Software verbessert. [4]

präventive Wartung
Bei der präventiven Wartung werden proaktiv Fehler gesucht, welche noch nicht im laufenden Betrieb aufgefallen sind. Diese Tätigkeit ist folglich planbar. [4]

Bommer et Al. [4] kategorisiert diese zudem nach einer reaktiven oder proaktiven Beteiligung. Dies wird in Grafik 3.2 gezeigt. Darüber hinaus steigt der Schwierigkeitsgrad einer Wartung zunehmend. Denn das Ziel dabei ist es die Software Funktionalitäten erneut an die Business Needs anzupassen. Strukturelle Verbesserungen und Fehlerbehebungen genügen dafür, bei einer stark gewachsenen Software, nicht mehr.

	Korrektur	Verbesserung
Reaktiv	Korrektive Wartung (Restfehler)	Adaptive Wartung (lack of movement)
Proaktiv	Präventive Wartung (Restfehler)	Perfektionierende Wartung (ignorant surgery)

Abbildung 3.2: Art der Software-Wartung nach proaktiv und reaktiver Beteiligung [4]

3.3 Software-Modernisierung

Die Software-Modernisierung ist der Prozess der Weiterentwicklung einer Software. Dies geschieht, indem Teile ersetzt, neu-entwickelt oder auf eine neue Plattform migriert werden. [19] Khadka et Al. [25] nennt die zu modernisierenden Systeme „[...] Legacy Systems [...]“, sofern diese schwer anzupassen sind und dennoch kritisch sind für das laufende Geschäft. Die Software-Modernisierung passt Legacy-Systeme an gewachsene System-Anforderungen an, wenn dies nicht mehr durch Wartungsmaßnahmen getan werden kann. Daraus ergibt sich, dass die Software-Modernisierung dann eingesetzt wird, wenn eine Wartung zu schwierig und kostspielig ist.
Die Software-Modernisierung arbeitet dabei größere Änderungen in das System ein. Zu beachten ist allerdings, dass ein Großteil des Systems beibehalten wird. Dies ist dann von Nutzen, wenn ein Software-System Anpassungen benötigt. Allerdings der Kern des Systems weiterhin ein Business Value hat. [26]
Seastorm et Al. [26] unterscheidet zudem *White-Box Modernisierung* und *Blackbox Modernisierung*. Die White-Box Modernisierung benötigt wissen über die internen Komponenten. Wohingegen Black-Box Modernisierung nur Kenntnisse über externe Schnittpunkte benötigt.

3.4 Software-Neuentwicklung

Die Neuentwicklung benötigt den Aufbau eines komplett neuen Software-Systems. Folglich ist dieser Ansatz kosten- und zeit-intensiv. Die Software-Neuentwicklung bietet sich an, wenn sowohl White-Box, als auch Black-Box Modernisierung nicht kostengünstiger als die Neuentwicklung sind.
Hier ist anzumerken, dass nicht alle Bestandteile des Systems gleichzeitig erneuert werden müssen. Eine Architektur bestehend aus klar abgetrennten Systemen kann auch teilweise neu entwickelt werden.
In diesem Stadium gilt zu evaluieren, ob eine kommerzielle Software für die gegebenen Anforderungen geeignet ist. Diese sogenannte Commercial off-the-shelf (COTS) Software [1] sind in unter anderem als Customer-Resource-Management-Systeme oder Finanz-Systeme verfügbar. COTS Software-Systeme sind bereits fertig implementierte Software-Bausteine für die genannten Anwendungsfälle. Eine firmeneigene Entwicklung kann so vermieden werden. [1][26]

4 Refactoring

In diesem Kapitel wird das Refactoring näher beleuchtet. Refactoring stellt eine Möglichkeit dar, die Alterung einer Software zu verlangsamen. Zunächst wird dabei der Begriff Refactoring erläutert. Mit dem gesammelten Wissen können anschließend einige konkrete Refactoring Maßnahmen beleuchtet werden. Dabei wird sich auf das Buch Refactoring [11] von Martin Fowler und Kent Beck bezogen. Anschließend wird das Refactoring mit der Software-Modernisierung in Verbindung gesetzt. Das Kapitel schließt mit der Frage, wann Refactoring-Maßnahmen angewendet werden sollten.

4.1 Was ist Refactoring

Mit dem Begriff Refactoring kann sowohl das Substantiv, wie auch das Verb betrachtet werden. Grundsätzlich findet bei dem Refactoring eine Änderung an der internen Struktur von Software statt. Beispielsweise kann das innere Verhalten effizienter sein. Auch besser lesbarer Code oder die Bündelung von zuvor verstreuten Verhalten sind Beispiele für Refactorings. Laut Flower und Beck ist das Ziel des Refactorings stets den Code einfacher verständlich und billiger modifizierbar zu machen. Diese Änderungen sind von außen nicht wahrnehmbar. Das äußere Verhalten bleibt unverändert.
Das Ziel dieser Änderungen ist, dass er Code leichter lesbar und zu modifizierbar wird. Da sowohl lesen, als auch modifizieren weniger Zeit in Anspruch nehmen, werden diese billiger.
Allerdings führen die internen Änderungen des Refactorings zu keiner Veränderung des äußeren Verhaltens der Software. Das Verb Refactoring meint nun die Anwendung einer Reihe von konkreten Refactorings. Der Autor und Informatiker Martin Fowler und der Informatiker Kent Beck haben in ihrem Buch Refactoring eine Reihe dieser konkreten Refactorings definiert. [11]

Refactoring sollte dabei nicht als Synonym für jede Art von Code-Restrukturierung und Bereinigung verwendet werden. Flower und Beck sehen in Refactoring eine sukzessive Anwendung von einzelnen Refactorings.[11] Jeder dieser Schritte erhält das Verhalten der Anwendung. Der Code bleibt daher nach jedem einzelnen Refactoring ausführbar mit dem identischen Verhalten wie zuvor. Refactoring ist eine besondere Art der Restrukturierung.

4.2 Bad-Smells

Um Kandidaten für Refactoring-Maßnahmen zu finden, muss der Begriff Bad-Smell betrachtet werden. Der Autor und Entwickler Martin Flower definiert ein Bad-Smell als einen oberflächlichen Hinweis, der in der Regel mit einem tieferen Problem im System korrespondiert. [23]
Martin Flower und der Entwickler Kent Beck begründen die Wahl Wortes *Smell* damit, dass dieser schnell zu erfassen ist. Ein Mensch kann einen Geruch wahrnehmen, ohne darüber nachzudenken. Ferner muss nicht klar sein, wo die Ursache des Geruches ist.

Man weiß nur, dass man etwas riecht.
Martin Flower nennt hier als Beispiel Methoden mit überdurchschnittlich vielen Parametern. Letzteres fällt einem aufmerksamen Entwickler ins Auge, ohne das er aktiv darüber nachdenkt. Ein weiterer Grund für die Wahl des Wortes Smell war, dass ein Geruch nicht zwangsläufig etwas Schlechtes ist. Methoden mit überdurchschnittlich vielen Parametern können gerechtfertigt sein. Der Bad-Smell kann sich folglich auch als ungefährlich herausstellen.
Tools wie zum Beispiel *SonarQube* [27] helfen dem Entwickler dabei Bad-Smells zu finden. Anschließend können Refactoring-Maßnahmen angewendet werden.

4.3 Refactorings und die Software-Modernisierung

Die grundsätzliche Aufgabe von Refactorings ist es Lehmans Gesetzen entgegenzuwirken. Konkret soll der Verfall der Architektur verlangsamt werden. Wird eine Architektur zunehmend modifiziert, hat dies eine kumulative Wirkung. Umso härter es ist, das Design im Code zu sehen, umso härter ist es dieses zu erhalten. Folglich verfällt das eigentliche Design zunehmend schneller. Regelmäßiges Refactoring hilft dabei den Code sauber zu halten. Ferner soll dem Wachstum der Code-Basis entgegengewirkt werden. [11][7]

Fowler stellt in seinem Buch Refactoring [11] einige konkrete Refactoring Maßnahmen vor, mit welchem Code Vervielfältigung entgegengewirkt wird. Primär ist das Ziel jede Deklaration von Verhalten genau ein Mal und auch nur ein Mal zu definieren. Mittels Refactoring werden etwaige doppelte Deklarationen gefunden und auf eine Deklaration reduziert. Oft fallen solche Zusammenhänge nicht während der Entwicklung auf. Das Refactoring hilft dabei, dies nachzuholen.

Darüber hinaus definiert Kent Beck die Design *Stamina Hypothesis* [11]. Diese Hypothese beschreibt, dass indem wir uns um ein gutes internes Design bemühen, wir länger schneller arbeiten können. Grafik 4.1 verdeutlicht diesen Zusammenhang. Die blaue Linie wächst zu Beginn schneller. Denn sie nutzt ein wenig durchdachtes Design. Dadurch können schnell Fortschritte erreicht werden, allerdings wird die Codebasis zunehmend schwieriger anzupassen. Die rote Kurve nutzt ein durchdachtes Design. Folglich wächst sie zu Beginn langsamer, anschließend allerdings stetig. Zuvor ein perfektes Software-Design zu erstellen, ist unwahrscheinlich. Daher wird Refactoring essenziell, um dieses während der Entwicklung zu erreichen.

Abschließend hilft Refactoring bei der Fehlersuche. Während der Einarbeitung von Refactorings beschäftigt sich der Entwickler eingehend mit bereits geschriebenen Code. Dadurch wird das Verständnis für den zugehörigen Code erweitert. Dieses erweiterte Verständnis kann mittels weiteren Refactoring-Maßnahmen in den Code einfließen. Dadurch können Annahmen und Überlegungen des Entwicklers klarer im Code selbst repräsentiert werden. Die klare Repräsentation von Annahmen erleichtert anschließend die Fehlersuche. Zusammenfassend lässt sich sagen, dass Refactoring-Maßnahmen dabei helfen Lehmanns Gesetzen entgegenzuwirken. Der bereits bestehende Code wird klarer formuliert und das Verständnis für den Code wird vertieft. Dadurch können Fehler schneller gefunden werden und dem Verfall der Architektur wird entgegengewirkt.

4.4 Software-Modernisierung im Vorgehensmodell

Mit dem gesammelten Wissen stellt sich die Frage, wann konkret Refactoring-Maßnahmen angewendet werden sollten.
Martin Fowler beschreibt dies wie folgt: "Refactoring is something I do every hour I program.". Darüber hinaus nennt Fowler "The best time to refactor is just before I need to add a new feature to the code base.". [11]
Der Entwickler Daniel Krämer beschreibt in dem Magazin Objektspektrum vom Mai 2020 ein konkretes Vorgehen für Refactorings innerhalb einer laufenden Entwicklung. Er sieht es ebenfalls als ratsam an, bereits während der Planung eine Vorstellung vom Umfang der beabsichtigten Aufräumarbeiten zu gewinnen und deren Kosten explizit in die Schätzung eines Tasks einfließen zu lassen.
Daraus ergibt sich, dass Refactoring und das Implementieren von neuen Funktionen, zwei voneinander getrennte Aufgabenbereiche sind. Während der Planung sollten beide Aufgabenbereiche eine Rolle spielen und beiden sollte genügend Zeit eingeplant werden. Darüber hinaus erleichtern das Schreiben von Regressionstest die Arbeit mit Refactorings. [7][11]

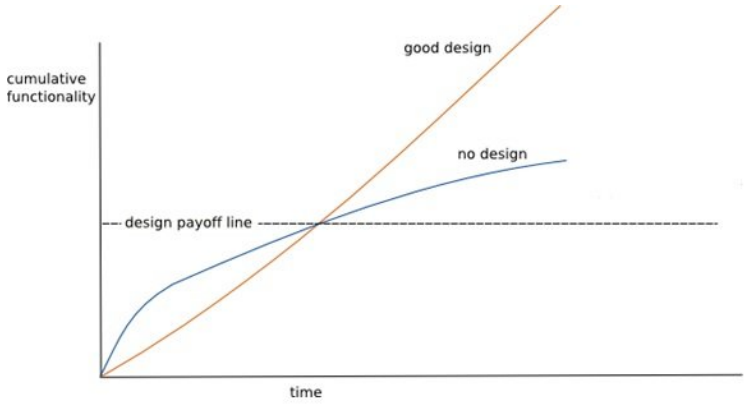


Abbildung 4.1: Design Stamina Hypothesis [11]

5 Vorgehensmodell für Cloud Computing

5.1 Vorab

Auf den in den vorigen Teilen beschriebenen Grundlagen, Techniken und Vorteilen von Software-Modernisierung aufbauend wird folgend mit Cloud Computing ein Praxisbeispiel behandelt. Abhängig der jeweiligen Branche greifen heutzutage immer mehr Unternehmen auf Cloud Lösungen von Drittanbietern(Provider) zurück. Diese bieten den Unternehemen, neben der Angebotsvielfalt, einige qualitative und quantitative Benefits. Daher erfreut sich deren Verwendung heutzutage immer größer werdender popularität und Beliebtheit [3]. Die Entscheidung der Betrachtung für Cloud Computing in dieser Ausarbeitung erfolgte nach keinem Ranking, das Cloud Computing im Vergleich von anderen Technologien hervorhebt. Der Einsatz dieser Lösung in der Praxis richtet sich nach den Bedürfnissen und Zielen der Unternehmen. Für die weiteren Betrachtungen wird von Unternehmen in der freien Marktwirtschaft ausgegangen.

Als exemplarische, moderne, zukunftsorientierte Technologie liefert Cloud Computing Grund genug eine facettenreiche Betrachtung einer Software-Modernisierungsmaßnahme durchzuführen. Die entstehenden zahlreichen Fragestellung, Analysen, Auswertungen und Evaluationen beeinflussen maßgeblich die zukünftige IT-Landschaft eines Unternehmens.

Eine Modernisierungsmaßnahme, bei der ein Legacy-System zu einer künftigen Cloud-Lösung entwickelt werden soll, erfordert gewisse Grundlagen. Neben der Definition des *National Institut of Standards and Technologies* zu Cloud Computing, wird notwendige Wissen in Abschnitt 5.2 Cloud Computing im Überblick vermittelt. Anschließend wird in Kapitel 5.3 On-premise in-hose vs Cloud Computing bei Providern die Lösungsmöglichkeit von einer „on-premise in house-Lösung“ zu Cloud Computing abgegrenzt. Die konkrete Durchführung einer Software-Modernisierungsmaßnahme stellt für die Unternehmen ein kompliziertes und komplexes Projekt dar. Zur Weiteren Vermittlung wird in Abschnitt 5.4 Vorgehensmodell zur Umsetzung für Cloud Computing eine vereinfachte Roadmap, also ein Vorgehensmodell zur Umsetzung einer Modernisierungsmaßnahme einer Cloud Computing Lösung vorgestellt. Die Möglichkeiten wie Services, beziehungsweise Dienste, in der späteren Cloud angeboten werden können, wird in 5.5 Schichtenmodell des Cloud Computings voneinander abgrenzt. In 5.6 Bekannte Cloud-Computing Anbieter schließt sich der Kreis, mit der abschließenden Vorstellungen von Amazons „Amazon web services (aws)“ und Microsofts „Azure“.

5.2 Cloud Computing im Überblick

Für einen gemeinsamen Kontext zu dem Begriff Cloud Computing wird die Definition des National Institut of Standards und Technology im Rahmen dieser Ausarbeitung verwendet. Dieses führt fünf grundlegende und einheitliche Charakterstika für Cloud Computing Systeme ein. Die zusammenfassenden Oberbegriffe dieser sind der On-demand self service, Broad network access, Resource pooling, die Rapid elasticity und dem Measured service.

Im Wesentlichen fassen die genannten Punkte die Möglichkeiten zusammen für einen entfernten, einseitigen Zugriff aus dem Netz mit heterogenen Endgeräten auf Rechen- und Serverressourcen. Diese Ressourcen werden, in Abhängigkeit der Bedürfnisse und Notwendigkeit, skalierbar und bedarfsgerecht, nach dem Mehrmandantenprinzip bereitgestellt. Die Benutzer und Benutzerinnen sehen dabei nicht, mit welchen (physischen) Kapazitäten die Leistung erbracht wird. Der measured Service beschreibt die Anforderung für die Bereitstellung erhobener Daten. Diese schließen unter anderem Kennzahlen wie die Zugriffszeiten, Anzahl aktiver Nutzer und Nutzerinnen, Server- und Speicherauslastung erhöhen die Transparenz mit ein. Dies ist nicht nur für die Transparenz seitens der Unternehmen wichtig, die für diese Dienste bezahlen, sondern auch für die Drittanbieter dieser Lösungen, die somit ihre Ressourcen überwachen und gegebenenfalls mit physischer Hardware der erwartbaren Nachfrage aufrüsten müssen. [28]

Cloud Computing basiert auf der **SOA!** (**SOA!**) [14], also der Service-, beziehungsweise Dienst-Orientierten Architektur. Ein Dienst/Service ist eine in sich geschlossene Softwareeinheit, die mit Anwendungen und/oder anderen Diensten über einen lose gekoppelten, oft asynchronen Kommunikationskanal kommuniziert. **SOA!** bezeichnet dabei die gesamte Architektur, bestehend aus den miteinander kommunizierenden Diensten. Grundlegende Eigenschaften der **SOA!** ist ein einheitlicher Kommunikationskanal und die lose gekoppelten Dienste. Bei deren Entwicklung liegt der Fokus auf einem qualitativ hochwertigen, bereitgestellten Dienst. [22]

Dies ist nicht ohnehin von besonderer Bedeutung, da man Fehler aus der Vergangenheit (Beispielweise. schlechte Code-Entwicklung, schlechtere Services, etc.) nicht im neuen System wiederholen möchte. **SOA!** bietet somit in seinen Grundzügen einen soliden Kern, auf dem Cloud Computing aufbaut.

Neben der Bestimmung der quantitativen und qualitativen Vor- und Nachteile, spielen oft weitere Faktoren eine wichtige Rolle. Auch die Auswahl von der Unternehmensgröße (aus Kostengründen) und der Branche des Unternehmens spielt eine wichtige Rolle. Beispielsweise unterliegen Behörden oder Betrieben der Sektoren Banken, Versicherungen oder sonstigen Finanzdienstleister besonderen (Datensicherheits-)Gesetzen. Für die Einhaltung von (IT-)Compliance oder etwaigen internen Kontrollsystemen, kommt eine on-premise und in-house Lösung für die Modernisierungsmaßnahme, beziehungsweise der zukünftigen IT-Landschaft infrage. [9]

5.3 On-premise in-hose vs Cloud Computing bei Providern

Um auch Cloud Computing, von einer im Unternehmen entwickelten und betriebenen Software zu unterscheiden (On-premise in-house), werden im Folgenden die Punkte Deployment, Kontrolle, (Daten-)Sicherheit und (IT-)Compliance Anforderungen voneinander abgegrenzt. Es soll deutlich gemacht werden, welche Vor- und Nachteile die Unternehmen im Zuge der Überlegungen für die Software-Modernisierung betrachten sollen und welche Restriktionen beeinflussen. On-Premise Software ist auf den Unternehmensservern installiert. Der Zugriff ist, neben der Firewall, von verschiedene in-house Faktoren abhängig. Diese Applikationen gelten als zuverlässig und sicher und ermöglichen den Unternehmen, ein hohes Maß an Kontrolle. [10]

Folgend werden die beiden Lösungsvarianten unter verschiedenen Gesichtspunkten voneinander abgegrenzt.

Deplyoment

In einer on-premise Enviroment findet das Deployment innerhalb der Unternehmen-IT-Infrastruktur statt. Entsprechend sind die Verantwortlich für die Wartung der Lösung und alle mit ihr verbundenen Prozesse.
Die virtuelle Cloud-Infrastruktur hingegen bietet hohe Flexibilität bei der Implementation auf einer breiten Infrastruktur. Zusätzlich ermöglicht es eine schnellere Installation und Support-Services, was auch die Systemadministrator effizient unterstützen kann.[2]

Kontrolle

Die Unternehmen mit einer on-premise Lösungen erheben, verfügen, analysieren und evaluieren die Unternehmen alle Daten. Was besonders für die Verwendung einer on-premise Lösung für Unternehmen spricht, die in einer stark-regulierten Branche arbeiten.
In der Cloud-Computing-Umgebung stellen sich viele Unternehmen und Provider die Frage nach des Eigentums der Daten. Diese befinden sich physisch bei den Drittanbietern. Falls es zu Problemen und Ausfällen kommt, können Unternehmen möglicherweise nicht auf diese Daten zugreifen.[2]

(Daten-)Sicherheit

Unternehmen mit besonders sensiblen Informationen, wie Behörden und Banken, müssen über ein gewisses Maß an Sicherheit und Datenschutz verfügen, das eine on-premise in-house Lösung bietet. Sicherheitsbedenken sind im Allgemeinen auch bei durch Drittanbieter angebotenen Cloud Computing-Lösungen ein Problem. Auch, wenn die Provider im Laufe der Zeit die Robustheit der gesamten Infrastruktur unter Beweis gestellt haben. [20] [9]

Compliance

Viele Unternehmen arbeiten, unabhängig von der Branche, unter behördlichen Kontrollen. Für Unternehmen, die solchen Vorschriften unterliegen, ist es zwingend erforderlich, dass sie ihre (IT-)Compliance permanent im Auge haben. Hierfür eignet sich die on-premise Variante eher.
Auch dann sind Unternehmen für die Einhaltung von (IT-)Compliance Richtlinien verantwortlich, wenn sie eine Cloud-Lösung bei einem Drittanbieter haben. Sensible Daten müssen geschützt werden um Kunden/Kundinnen, Partner/Partnerinnen und Mitarbeiter/Mitarbeiterinnen ihre Privatsphäre gewährleisten zu können.[20]

5.4 Vorgehensmodell zur Umsetzung für Cloud Computing

[With legacy systems] the cost is getting higher because maintenance is getting more expensive, [then] maybe you should think of modernization — unbekannt [18]

Die Entwicklung, der Betrieb, die Betreuung und die stetige Weiterentwicklung durch neue Anforderungen von Software im Unternehmen erfordert mehr Expert:innenwissen als früher. Ursache dafür ist auch der immer größer werdende Anteil künstlicher Intelligenz in Applikationen. Es wird Hardware benötigt. Ebenso weitere physischer Computerressourcen für Fallback-Szenarien, die auf die Gewährleistung von (IT-)Compliance einzahlen.

Repräsentative und auswertbare Studien bereits umgesetzter Modernisierungen größerer Unternehmen finden sich in der Literatur nur wenige. Etwaige Gründe dafür könnten sein, dass diverse erfolgreiche Lösungen aufgrund des Konkurrenzdenkens und resultierenden Marktvorteil nicht veröffentlicht werden. Obwohl immer mehr Unternehmen die Bedeutung und die Vorteile von signifikant besser wartbarer und skalierbarer Software erkennen, führen sie keine Modernisierungsmaßnahmen durch. Eine mögliche Erklärung dafür kann an dem mangelnden Fachpersonal liegen.

Trotz einer gut definierten Enterprise-Architektur und unmissverständlich dokumentierten Sourcecode (Refactoring) einer Legacy-Software, begegnen dem (IT-)Personal viele Hürden. Der Weg der Legacy Software bis hin zur „modernen Software“ muss klar definiert, strukturiert und deterministisch sein.

Die Software-Modernisierung einer Legacy ist keineswegs trivial. Zahlreiche Herausforderungen, die nicht nur Geld kosten, erschweren den Weg von der Legacy zur modernen Software. Wie kann die zukünftige Software qualitativ verbessert werden? Was gilt überhaupt zu beachten? Welche Anforderungen galten bisher? Wie soll die Ziellandschaft aussehen? Eine ausgeklügelte Strategie kann für den langfristigen Erfolg und Erhalt der Software sorgen. Die Unternehmen benötigen eine Roadmap/ ein Vorgehensmodell. Es soll alle notwendige Schritte definieren, alle Fragestellungen kategorisch zusammenfassen und die Unternehmen in jedem Schritt der Modernisierungsmaßnahme unterstützen.

In der Literatur gibt es diverse Vorgehensmodelle zur Umsetzung von Modernisierungsmaßnahmen. Die Vielfalt dieser liegt unter anderem in den unterschiedlichen Anforderungen, verfügbaren Ressourcen, dem Risiko-Management, der zu modernisierenden Software und auch mit der mittel- bis langfristigen Unternehmensstrategie. [16]

5.4.1 ARTIST EU

Ein Modell-getriebener Ansatz Europäischen Union nennt sich ***A**lternative **R**outes **T**owards **I**nformation **S**torage and **T**ransport (ARTIST)*.[6] Die ARTIST Methoden und ARTIST Frameworks, sollen die Entwickler und Entwicklerinnen in jedem Schritt der Migration- und Modernisierungsmaßnahme unterstützen. Dabei fasst es bereits Best-practises zusammen[24]. In Teilen verwendet ARTIST die **S**ervice-**O**riented **M**igration and **R**euse **T**echnique (SMART) die in der Pre-Migration-Phase stattfindet.

SMART wird unter anderem auch für die Analyse der Legacy verwendet, um festzustellen, ob beispielsweise ihre Funktionalität oder Teile davon als Services (**SOA!**) be-

reitgestellt werden können. Die Interaktion eines Legacy-Systems innerhalb einer **SOA!**, wie beispielsweise einer Web-Services-Architektur, ist manchmal relativ einfach – was wiederum sehr attraktiv für die Unternehmen ist. Web-Service-Schnittstellen sind so eingerichtet, dass sie SOAP-Nachrichten empfangen und analysiert werden, wodurch der entsprechende Dienst direkt aufgerufen werden kann. Zahlreiche neue Entwicklungsumgebungen bieten Tools, die diesen Prozess unterstützen. Unternehmen setzen eben diese Umgebungen ein, um ihre Geschäftsprozesse der Welt zugänglich zu machen.[22]

5.4.2 Roadmap

Das Vorgehensmodell zur Modernisierung einer Legacy von S. Jain und I. Chana [17] wird nun betrachtet. Einen groben Überblick bietet folgende Abbildung, die die wesentlichen Hauptkomponenten des Vorgehensmodells darstellt.

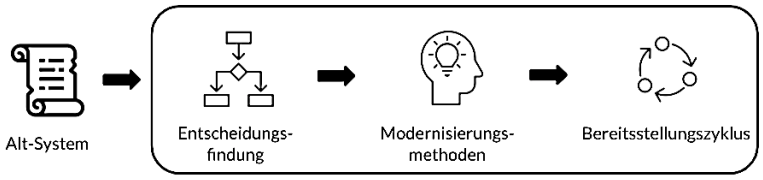


Abbildung 5.1: Vorgehensmodell zur Umsetzung einer Modernisierungsmaßnahme für Cloud Computing. [17]

Erstens, das Entscheidungsmodul. Wieder rum besteht aus drei Hauptkomponenten, die im Wesentlich zur Bewertung des Alt- und Zielsystem eine Grundlage bilden.
Zweitens, das Modul für Modernisierungsmethoden. Hier stehen uns vier Schlüsselmethoden zur Auswahl, um die Modernisierungsmaßnahme mit einem geeigneten Ansatz und einer geeigneten Implementierung fortzusetzen. Drittens, der Bereitstellungszyklus. Er beschreibt nach den ersten beiden Modulen einen permanenten Zyklus, bestehend aus vier zusammenfassenden Schritten.

Entscheidungsfindung

Das erste Modul kann im Weiteren in drei weitere Module unterteilt werden.

Beurteilung des Altsystems
: Die Bestimmung der Komplexität für die Modernisierungsmaßnahme ist ein Hauptfaktor um den Aufwand und die Kosten für das Projekt bestimmen zu können. Dabei müssen die unterschiedlichen Schichten der Software, oder der IT-Infrastruktur betrachtet werden. Verglichen mit dem OSI-7-Layer Modell müssen die sieben Schichten Physical-, Data Link-, Network-, Transport-, Session-, Presentation-, Session- und Application-Layer untersucht werden [5]. Hier sollte bestimmt werden, in welchen Schichten beispielsweise die Business Logik liegt und mit welchen anderen Layern sie korrespondiert. Die Komponenten, die in das Cloud-System migriert werden sollen, beeinflussen somit die Skalierbarkeit der Applikationen in.

Beurteilung (bisherige) Servicequalität
: Ebenfalls muss die bisherige Qualität analysiert werden. Was war bisher schlecht und was könnte im Zuge der Modernisierung besser gemacht. Aus technischer

sowie auch fachlicher Sicht. Konkret bedeutet das, ob man beispielsweise durch ein Redesign des Prozesses größere qualitative als auch quantitative Erfolge erzielen kann. Das könnten beispielsweise verkürzte Arbeitswege sein.

Bestimmung von Zielen
Aus den bisherigen Betrachtungen sollten nun die Ziele bestimmt werden. Das könnte auf allgemeinen Ebene z.B. sein: Ein besseres Management der Plattform, der Infrastruktur oder der Services. In Hinblick auf den Code können eine bessere Wartbarkeit, eine bessere Codequalität und quantitativ verringerte Komplexität von Funktion ebenso weitere Ziele sein.

Medornisierungsmethoden

S. Jain et al. stellen in ihrem Vorgehensmodell folgende vier Schlüsselmethoden vor

Ersetzung
Denkbar ist eine komplette Ersetzung von Software-Komponenten durch bereits existierende Software von Drittanbietern. COTS [12] gehen mit einem geringeren Risiko einher, da die Software bereits etabliert und getestet ist. Oftmals ist aber kein direkter Kauf mehr möglich, da viele Drittanbieter ihre Lösungen über diverse Lizenzmodelle anbieten. Das führt zu dem Nachteil, dass ein Unternehmen mittel- bis langfristig in hohem Maße von dem Anbieter abhängig sind.
Ebenfalls kommt dieser Ansatz nicht infrage, sofern die bewertete Business-Logik sehr an die Bedürfnisse der Unternehmen angepasst wurde und damit nicht hundertprozentig mehr mit COTS ersetzbar ist.

Black Box Wrapping
In diesem Ansatz wird ein neues Interface lediglich an alte Komponenten der Legacy in der Cloud-Umgebung angebunden. Umsetzbar ist dieser Ansatz nur dann, wenn der Code der Legacy in einem gut-verwertbaren Zustand ist (vgl. Refactoring). Ebenfalls beinhaltet dieser Ansatz eine sukzessive Anpassung der noch bestehenden Altsoftware im neuen System.

Reengineering/Redevelopment
Übergreifend mit dem Begriff Software-Engineering wird hier verwendet. Dieser Ansatz stellt somit eine komplette Neuentwicklung des Systems dar, mit teilweise Wiederverwendung oder Verbesserungen des alten Codes.

Migration
Das komplette System mit Kernfunktionalität wird in eine neue Umgebung repliziert, mittels unterschiedlicher Migrationsstrategien. So können auch Altsysteme bestehend und ohne Veränderung auf einer virtuellen Umgebung kurz- bis mittelfristig geparkt werden. Anschließend kann eine sukzessive Anpassung der Software erfolgen. An dieser Stelle erwähnenswert sind die Arten, wie spätere Dienste oder Applikationen in der Cloud angeboten und dargestellt werden können. Konkret handelt es sich hierbei um Infrastructe as a Service (IaaS), Process as a Service (PaaS) und SaaS. Diese Begriffe werden in Kapitel *5.5 Schichtenmodell des Cloud Computings* näher behandelt.

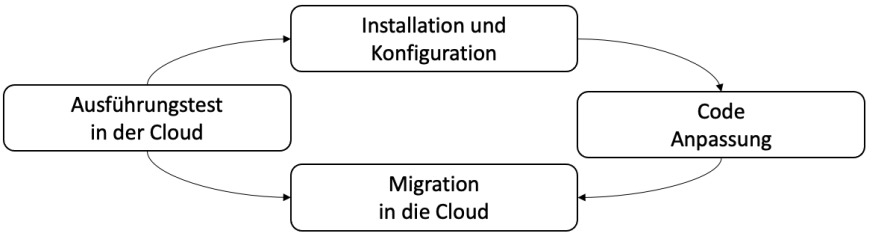


Abbildung 5.2: Bereitstellungszyklus im Vorgehensmodell [17]

Bereitstellungszyklus

Nach der Selektion des geeigneten Ansatzes wird inkrementell mit der Implementierung begonnen, um die Modifikationen schrittweise mit den betroffenen Bestandteilen abzubilden. Mit der ersten Phase beginnend, kommt die Einrichtung und Einstellung der nötigen Tools für die Migration. Einzelne Code-Segmente werden in den zirkulierenden Phasen priorisiert angepasst. Es muss stets ein Fallback-Szenario vorhanden sein, bevor Änderungen am Quellcode durchgeführt werden. Hier kann ein Replikat der funktionierenden Alt-Version verwendet. Die modifizierte Komponente wird dann in der Cloud mit ausgewählten Servicemodellen (vgl. *5.5 Schichtenmodell des Cloud Computings*) und den jeweiligen Bereitstellungskonfigurationen gehostet. Abschließend findet in jedem Zyklus ein vollumfänglicher Test der Funktionalität innerhalb der Cloud statt, damit das System auch stets funktionsfähig bleibt und korrekt arbeitet.

5.5 Schichtenmodell des Cloud Computings

Die Art und Weise, wie eine Software über ein Cloud-System bereitgestellt wird, ist Teil der Überlegungen für die Bestimmung der Ziele und Ziellandschaft. Cloud Computing unterteile dabei in drei Schichten, die Folgendermaßen erläutert werden. Es werden die Definitionen der einzelnen Schichten von IBM verwendet [13].

Infrastructe as a Service
: Hier bieten die Service-Provider Cloud-Lösungen, wie etwa Speicher, Netzbetrieb oder Server bereit. Dies bietet den Unternehmen den Vorteil Kosten für den Kauf und die Wartung eigener Hardware zu sparen. Und da die Daten alle in der Cloud liegen, gibt es keinen Single point of failure. Der Single point of failure ist im Prinzip eine Fehlfunktion, die zum Ausfall des Systems führen kann.

Process as a Service
: Hier wird den Usern eine komplette Cloud-Umgebung bereitgestellt. Zusätzlich zu dem Speicher und IT-Ressourcen bekommen die Nutzer eine Reihe vordefinierter Tools an die Hand. Die ist insbesondere interessant für Tests und Entwicklungen.

Software as a Service
: Hier wird eine Anwendungen Cloud-basiert angeboten. Der Zugriff erfolgt meist über die Programmschnittstellen oder Webinterfaces. Damit kann an über, beispielsweise den Webbrowser auf Applikationen des eigenen Unternehmens entfernt zugreifen, ohne sich in einem Virtuellen privaten Netzwerk zu befinden.

5.6 Bekannte Cloud-Computing Anbieter

Die Entscheidung bei welchem Anbieter Unternehmen ihre Cloud-Lösungen beziehen wollen, spielt eine wichtige Rolle im Modernisierungsprozess. Technik-Giganten wie Amazon mit *Amazon Web Services (AWS)*, Microsoft mit *Azure* und Google mit *Google Cloud* bieten mittlerweile schon länger Cloud-Lösungen für Unternehmen, als auch Privatpersonen an. Sie bieten alle die in Kapitel 5.5 Schichtenmodell des Cloud Computings vorgestellten Arten IaaS, PaaS, SaaS. Interessante Gegenüberstellungen wie Bedienbarkeit, Geschwindigkeit und Zuverlässigkeit lassen sich in der Literatur, als auch von den Anbietern keine finden. Die für Kunden merkbaren Unterschiede stellen daher nur die Anbieterspezifischen Zusatzvorteilen, wie zum Beispiel Tools, Funktionen und Werkzeuge dar.

5.7 Zusammenfassung

Den Unternehmen, die Lösungen bei Cloud-Anbietern beziehen wollen, stehen heute eine Fülle an verschiedenen Möglichkeiten und Angeboten zur Auswahl. Neben der Angebotsvielfalt verschiedenerer Anbieter für Cloud-Systeme gibt es bereits auch viele Tools und Werkzeuge, die die Software-Modernisierung kostengünstiger und effizienter gestalten können. Unterstützt mit diversen Vorgehensmodellen, gestaltet sich so die Modernisierungsmaßnahme als keinen mehr zu groß unüberwindbar-wirkenden Monolithen mehr. Somit ist es so einfach wie noch nie veraltete Software günstiger modernisieren zu lassen.

Literatur

[1] Chris Abts, M. S. Barry, Barry W. Boehm, Ph D. Elizabeth, Elizabeth Bailey Clark und Ph D. „COCOTS: A COTS Software Integration Lifecycle Cost Model - Model Overview and Preliminary Data Collection Findings“. In: *In ESCOM-SCOPE Conference*. Publ, 2000, S. 2000–2000.

[2] Stamatia Bibi, Dimitrios Katsaros und Panayiotis Bozanis. „Business application acquisition: On-premise or SaaS-based solutions?“ In: *IEEE software* 29.3 (2012), S. 86–93.

[3] Alexandra Boldyreva und Paul Grubbs. „Making encryption work in the cloud“. In: *Network Security* 2014.10 (2014), S. 8–10.

[4] Christoph Bommer, Markus Spindler und Volkert Barr. *Softwarewartung: Grundlagen, Management und Wartungstechniken*. Deutsch. 1. Edition. dpunkt.verlag, Apr. 2016.

[5] Neil Briscoe. „Understanding the OSI 7-layer model“. In: *PC Network Advisor* 120.2 (2000), S. 13–15.

[6] European Commision. *Alternative Routes Towards Information Storage and Transport at the Atomic and Molecular Scale*. 2013. URL: https://cordis.europa.eu/project/id/243421 (besucht am 18. 07. 2021).

[7] Daniel Krämer. „Legacy-Software wieder testbar machen: Jenseits der grünen Wiese“. Deutsch. In: *OBJEKTSpektrum - Software-Modernisierung* 05 (Mai 2020), S. 63.

[8] *Datenschutz-Grundverordnung: DSGVO als übersichtliche Seite*. de-DE. URL: https://dsgvo-gesetz.de/ (besucht am 30. 05. 2021).

[9] Michael Falk. *IT-Compliance in der Corporate Governance: Anforderungen und Umsetzung*. Springer-Verlag, 2012.

[10] Cameron Fisher u. a. „Cloud versus on-premise computing“. In: *American Journal of Industrial and Business Management* 8.09 (2018), S. 1991.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Nov. 2018.

[12] Herman Holtz. „Some Off-the-Shelf Systems“. In: *Computer Work Stations*. Springer, 1985, S. 15–24.

[13] IBM. *IaaS vs. PaaS vs. SaaS*. 2018. URL: https://www.ibm.com/de-de/cloud/learn/iaas-paas-saas (besucht am 09. 07. 2021).

[14] IBM. *What is SOA, or service-oriented architecture?* 2021. URL: https://www.ibm.com/cloud/learn/soa (besucht am 04. 07. 2021).

[15] Ayelet Israeli und Dror G. Feitelson. „The Linux kernel as a case study in software evolution“. In: *Journal of Systems and Software* 83.3 (März 2010), S. 485–501. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.09.042. URL: https://doi.org/10.1016/j.jss.2009.09.042 (besucht am 30. 05. 2021).

[16] Suman Jain und Inderveer Chana. „Modernization of Legacy Systems: A Generalised Roadmap“. In: *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*. ICCCT '15. Allahabad, India: Association for Computing Machinery, 2015, 62–67. ISBN: 9781450335522. DOI: 10.1145/2818567.2818579. URL: https://doi.org/10.1145/2818567.2818579.

[17] Suman Jain und Inderveer Chana. „Modernization of Legacy Systems: A Generalised Roadmap“. In: *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*. ICCCT '15. Allahabad, India: Association for Computing Machinery, 2015, 62–67. ISBN: 9781450335522. DOI: 10.1145/2818567.2818579. URL: https://doi.org/10.1145/2818567.2818579.

[18] Ravi Khadka, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen und Jurriaan Hage. „How Do Professionals Perceive Legacy Systems and Software Modernization?“ In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 36–47. ISBN: 9781450327565. DOI: 10.1145/2568225.2568318. URL: https://doi.org/10.1145/2568225.2568318.

[19] Ravi Khadka, Prajan Shrestha, Bart Klein, Amir Saeidi, Jurriaan Hage, Slinger Jansen, Edwin van Dis und Magiel Bruntink. „Does software modernization deliver what it aimed for? A post modernization analysis of five software modernization case studies“. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sep. 2015, S. 477–486. DOI: 10.1109/ICSM.2015.7332499.

[20] Michael Klotz und Dietrich-W Dorn. „IT-Compliance—Begriff, Umfang und relevante Regelwerke“. In: *HMD Praxis der Wirtschaftsinformatik* 45.5 (2008), S. 5–14.

[21] M. M. Lehman. „On understanding laws, evolution, and conservation in the large-program life cycle“. en. In: *Journal of Systems and Software* 1 (Jan. 1979), S. 213–221. ISSN: 0164-1212. DOI: 10.1016/0164-1212(79)90022-0. URL: https://www.sciencedirect.com/science/article/pii/0164121279900220 (besucht am 30. 05. 2021).

[22] Grace Lewis, Edwin Morris, Dennis Smith und Liam O'Brien. „Service-oriented migration and reuse technique (smart)“. In: *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*. IEEE. 2005, S. 222–229.

[23] Martin Fowler. *CodeSmell*. URL: https://martinfowler.com/bliki/CodeSmell.html (besucht am 12. 06. 2021).

[24] Andreas Menychtas, Kleopatra Konstanteli, Juncal Alonso, Leire Orue-Echevarria, Jesus Gorronogoitia, George Kousiouris, Christina Santzaridou, Hugo Bruneliere, Bram Pellens, Peter Stuer u. a. „Software modernization and cloudification using the ARTIST migration methodology and framework“. In: *Scalable Computing: Practice and Experience* 15.2 (2014), S. 131–152.

[25] Ravi Khadka. *How do professionals perceive legacy systems and software modernization? | Proceedings of the 36th International Conference on Software Engineering*. URL: https://dl.acm.org/doi/10.1145/2568225.2568318 (besucht am 30. 05. 2021).

[26] Robert Seacord, Peter Gordon, Daniel Plakosh, Grace Lewis und John Fuller. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Englisch. 1. Edition. Boston: Addison-Wesley Professional, Feb. 2003. ISBN: 978-0-321-11884-4.

[27] SonarQube. *Code Quality and Code Security | SonarQube*. en. URL: https://www.sonarqube.org/ (besucht am 08. 07. 2021).

[28] National Institute of Standards und Technology. *The NIST Definition of Cloud Computing*. 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800145.pdf (besucht am 04. 07. 2021).

[29] The Hamburg Commissioner for Data Protection and Freedom of Information. *35.3 Million Euro Fine for Data Protection Violations in H&M's Service Center*. Pressemitteilung. Hamburg, Okt. 2020, S. 2.

Abbildungsverzeichnis

Abkürzungsverzeichnis

DSGVO Europäische Datenschutzgrundverordnung

COTS Commercial off-the-shelf

ARTIST **A**lternative **R**outes **T**owards **I**nformation **S**torage and **T**ransport

SMART **S**ervice-**O**riented **M**igration and **R**euse **T**echnique

IaaS Infrastructe as a Service

PaaS Process as a Service

SaaS Software as a Service

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.1). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt