Aspekte von Software-Wiederverwendung

Julian Fietkau

16. März 2012

Seminararbeit im Modul

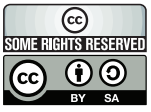
„Softwarearchitektur“

im Wintersemester 2011/12

Dozenten: Dr. Guido Gryczan, Dr. Carola Lilienthal

Fachbereich Informatik
Universität Hamburg

Inhaltsverzeichnis



Zusammenfassung

Trotz fortwährender Verbesserung der Werkzeuge bleibt die Neuentwicklung von Software ein kostspieliges Unterfangen. Aus ökonomischer wie auch pragmatischer Sicht liegt das Bestreben nahe, sie in möglichst hohem Maße wiederzuverwenden, anzupassen und in neue Kontexte einzubetten. Diese Arbeit gibt anhand von Kapitel 16 aus Ian Sommerville's Buch *Software Engineering* [Sommerville 2010] einen Überblick über diverse Strategien zur Software-Wiederverwendung, ergänzt dabei den Inhalt des Buches um den Themenkomplex *Open Source* und diskutiert die Wiederverwendung von Softwarekomponenten anhand von Steve Yegge's „Rant" über Google und Software-Plattformen.

1. Einleitung

Der Markt für Software ist heute größer denn je: Nicht nur, dass immer mehr Menschen diverse Computersysteme besitzen und damit potenzielle Nutzer für Anwendungssoftware sind, und dass nach wie vor immer mehr Firmen einen Bedarf für angepasste Software haben; hinzu kommt, dass auch eine stetig wachsende Anzahl von bestehenden Softwaresystemen gepflegt, modernisiert oder sogar vollständig ausgetauscht werden muss. Bei der Erstellung von Software aller Art liegt der Gedanke nahe, bestehenden Code wiederzuverwenden. Tatsächlich dürfte es heute kaum mehr möglich sein, Software zu schreiben, die nicht in irgendeiner Form auf bereits bestehenden Komponenten (und seien es Bibliotheken, Laufzeitumgebungen oder Frameworks) aufbaut. Entscheidend für die Frage, wie gut die verschiedenen Herangehensweisen an Software-Wiederverwendung ihren Zweck erfüllen, ist deshalb weniger die Frage „Wiederverwendung, ja/nein?" als vielmehr der Grad der Wiederverwendung – und damit der Zeit- und Kostenersparnis.

Dabei reichen die Möglichkeiten von der Wiederverwendung einzelner bewährter Code-Schnipsel über das Kopieren von bestehenden Klassen und Modulen bis hin zur mehrfachen Verwendung von kompletten Systemen in verschiedenen Umgebungen. Ebenso kann die Wiederverwendung technisch sehr unterschiedlich umgesetzt werden: Es könnte eine Komponente dupliziert und in eine neue Umgebung übertragen werden, es könnte stattdessen aber auch eine Verbindung zu einem vorhandenen System aufgebaut und diese eine Instanz auf diese Weise mehrfach verwendet werden. Es wird erkennbar, dass das Feld der Software-Wiederverwendung groß und in sich nicht leicht strukturierbar ist.

2. Zusammenfassung des Buchkapitels

In seinem Buch *Software Engineering* [Sommerville 2010] unternimmt Ian Sommerville den Versuch, einige Konzepte aus dem Gebiet der Software-Wiederverwendung exemplarisch ausführlich darzustellen. Dabei soll klar werden, welche Möglichkeiten sich durch unterschiedliche Ansätze auftun und wo ihre Grenzen liegen. Viele der Argumente finden sich wieder, wenn man andere als die genannten Herangehensweisen betrachtet.

Im Folgenden wird das Kapitel 16 zusammenfassend wiedergegeben, wobei seine Struktur übernommen wird um Quervergleiche zu vereinfachen. Sommerville untersucht dabei

im ersten Abschnitt übergreifende Konzepte und schafft einen Überblick, bevor er sich in den folgenden drei Abschnitten konkreten Beispielen zuwendet.

2.1. The Reuse Landscape

Um einen ersten Überblick zu ermöglichen und den Einstieg ins Thema zu erleichtern, werden zunächst drei Granularitätsstufen für Software-Wiederverwendung gesammelt:

1. **Wiederverwendung von Anwendungssystemen:** Ganze Anwendungen und bestehende Softwaresysteme werden wiederverwendet, entweder indem sie in unveränderter Form in eine Software-Umgebung eingebettet werden oder indem sie individuell angepasst werden und zum Beispiel Produktfamilien bilden.

2. **Wiederverwendung von Komponenten:** Subysteme und Komponenten von bestehender Software werden in neu entwickelte Softwaresysteme übernommen. In aller Regel sind dafür Anpassungen an den Schnittstellen nötig.

3. **Wiederverwendung von Objekten und Funktionen:** Einzelne Definitionen für Objekte und Funktionen mit klar definiertem Funktionsumfang können für eine Vielzahl konkreter Softwareprojekte eingesetzt werden, um allgemeine Funktionalität bereitzustellen.

Weiterhin werden eine ganze Reihe von allgemeinen Vor- und Nachteilen von Software-Wiederverwendung aufgezählt. Die einzelnen Methoden müssen sich an diesen Vorteilen und Problemen messen, denn sie gelten nicht alle für jede Methode im gleichen Maße.

2.1.1. Vorteile

Eine vermehrte Wiederverwendung von bewährter Software kann eine **erhöhte Verlässlichkeit** nach sich ziehen, da solche erprobte Software mit höherer Wahrscheinlichkeit bereits von gravierenden Fehlern befreit worden ist. (Dies ist natürlich keine Garantie, sollte im statistischen Mittel allerdings zutreffen.)

Aus ökonomischer Sicht ist ein **geringeres Prozessrisiko** ein nicht zu unterschätzender Vorteil. Damit ist gemeint, dass die Entwicklungskosten für bereits bestehende Software per Definition bereits erbracht sind. So gibt es weniger Unwägbarkeiten hinsichtlich der restlichen Entwicklung.

Wiederum ein gutes Argument aus Sicht der Kosten und Nutzen ist die **effektive Nutzung von Spezialisten**. Allgemeine und vielfach benötigte Lösungen existieren oft bereits in Form von Software, so dass die Entwickler sich den fachlichen Problemen widmen können. Auf diese Weise können sie Arbeit erledigen, die wirklich nur sie erbringen können und nicht jeder beliebige andere Entwickler.

Die Wiederverwendung von Software ist förderlich für die **Standardkonformität**. Somit kommen die Produkte besser mit der Herausforderung zurecht, rechtlichen und organisatorischen Anforderungen zu genügen und mit anderer Software fehlerfrei zu kommunizieren. Nicht selten entsteht aus einer weit genutzten Softwarelösung sogar ein neuer Standard.

Vor allem und nicht zuletzt führt die konsequente Durchführung und Nutzung von Software-Wiederverwendung auch insgesamt zu einer **schnelleren Entwicklung**, da wertvolle Entwicklungszeit für Kernkomponenten verwendet werden kann und die Entwicklung von bereits vorhandenen Teilen eingespart werden kann.

2.1.2. Probleme

Die Verwendung von bestehenden Softwarekomponenten kann auch **erhöhte Wartungskosten** bedeuten, etwa wenn nicht zu allen Softwarekomponenten der Quellcode vorliegt oder sie aus anderen Gründen nicht verändert werden können. Dann ist unter Umständen komplizierter Code zur Zugriffsverwaltung nötig, damit alle erforderlichen Schnittstellen zur Verfügung stehen.

Gelungene Software-Wiederverwendung kann in einigen Fällen durch **mangelnde Werkzeugunterstützung** erschwert werden, wenn die im Entwicklungsprozess verwendeten Werkzeuge keine Wiederverwendung vorsehen.

Bei dem **„Not-invented-here"-Syndrom** handelt es sich um eine humorvolle Umschreibung der Neigung vieler Softwareentwickler, fremdem Code nicht zu trauen. In vielen Fällen wird eine Softwarekomponente nur deshalb neu entwickelt, weil der Entwickler der Meinung ist, er könne bessere Arbeit leisten als alle vor ihm. Diese Haltung kann erfolgreiche Software-Wiederverwendung enorm erschweren, wenn zum Beispiel Fehler an falschen Stellen gesucht werden.

Für eine systematische Software-Wiederverwendung mit maximalem Nutzen ist früher oder später die **Pflege einer Komponentenbibliothek** nötig, in der wiederverwendbare Softwarekomponenten organisiert und katalogisiert werden. Dies muss im Entwicklungsprozess vorgesehen sein und kostet eigene Zeit.

Damit Hand in Hand geht die Anforderung, sich um **korrektes Sammeln und Verstehen von Komponenten** zu kümmern. Hier ist eher die Anforderung an die einzelnen Entwickler gemeint, die richtigen Komponenten zur Wiederverwendung selbst zu erstellen, aber auch auszuwählen.

2.1.3. Sammlung von Ansätzen

Ein Versuch einer groben Strukturierung von möglichen Ansätzen nimmt Ian Sommerville in seiner *Reuse Landscape* vor (siehe Abbildung 1, Seite 6). Dabei handelt es sich um eine lose Sammlung von Konzepten und Ansätzen, die im Umfeld der Software-Wiederverwendung wichtig sind. Darunter sind viele Begriffe, die in anderen Kapiteln des Buchs genauer betrachtet werden, und auch die Konzepte aus den noch folgenden Abschnitten dieses Kapitels tauchen dort auf.

Sommerville nimmt im Verlauf des Kapitels kaum mehr Bezug auf die Reuse Landscape, aber als Überblick über mögliche Vorgehensweisen und als Sammlung von Schlagworten für vertiefende Nachforschungen ist sie gut geeignet.

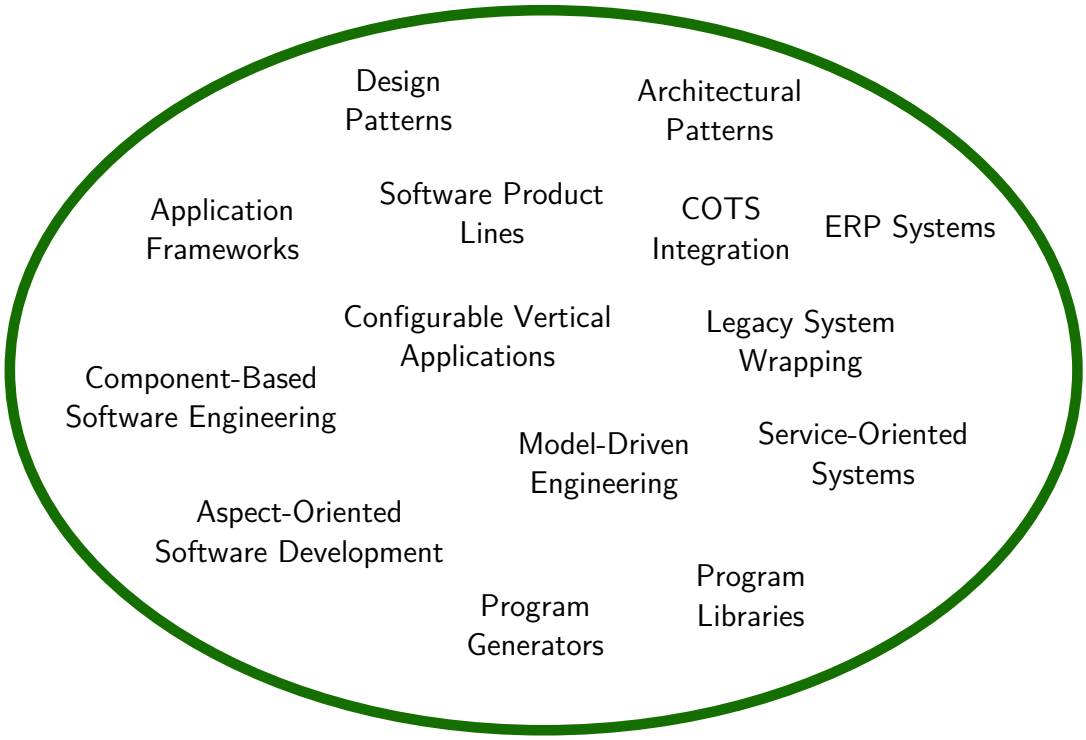


Abbildung 1: Ian Sommerville's *Reuse Landscape*. Hierin findet sich eine eher grob strukturierte Sammlung verschiedenster Konzepte zum Thema Software-Wiederverwendung. Einige verwandte Begriffe sind räumlich nahe angeordnet, aber insgesamt sind die genannten Ansätze sehr verschieden.

2.1.4. Schlüsselfaktoren bei der Wahl des Ansatzes

Nachdem die Breite der möglichen Ansätze aufgezeigt wurde, ist die Frage entscheidend, welche Faktoren bei der Wahl eines Ansatzes wichtig sind und nach welchen Kriterien entschieden werden sollte. Auch hierzu gibt Sommerville einige Stichwörter.

- **Zeitplanung:** Mit verstärkter Software-Wiederverwendung treten andere zeitliche Herausforderungen in den Vordergrund als wenn die Software von Grund auf neu entwickelt wird. In erster Linie kann Entwicklungszeit gespart werden. Inwieweit Komponenten neu entwickelt werden, muss sich daher auch nach der Zeitplanung des gesamten Projektes richten.
- **Geplante Lebenszeit der Software:** Die Art der Software-Wiederverwendung beeinflusst auf verschiedene Weisen die Verlässlichkeit und Wartungsfreundlichkeit der Software und damit ihre erwartete Lebenszeit. Praxiserprobte Komponenten können die Fehlerwahrscheinlichkeit senken, aber in manchen Fällen kann „Patchwork-Software“ sehr schwer wartbar sein.
- **Fähigkeiten der Entwickler:** Einige Methoden erfordern Fingerspitzengefühl und Erfahrung mit Software-Architektur, die nicht unbedingt alle Entwickler mitbringen. Allerdings gilt dies genau so für einige komplexe Probleme, wenn auf Software-Wiederverwendung verzichtet wird.
- **Nicht-funktionale Anforderungen:** Insbesondere Eigenschaften wie Stabilität und Performanz werden von der komponentenweisen Wiederverwendung in hohem Maße beeinflusst, was bei der Wahl von Laufzeitumgebungen, Frameworks und anderen relevanten Bestandteilen berücksichtigt werden muss.
- **Anwendungsfeld:** Der Einsatzort und -zweck der Software kann sich ebenfalls auf den Grad der möglichen Wiederverwendung auswirken, etwa durch technische Beschränkungen oder rechtliche Rahmenbedingungen.
- **Software-Plattform:** Mitunter gibt es bereits feste Vorgaben für die zugrundeliegende Plattform, auf der die Software laufen soll. Dabei kann es sich um bestimmte Geräte, Betriebssysteme oder auch Frameworks handeln. In aller Regel funktioniert die Wiederverwendung von Komponenten nur, wenn sie auf die gleiche zugrundeliegende Plattform ausgelegt sind.

2.2. Frameworks

Eine Form von Software-Wiederverwendung, die in den letzten ein bis zwei Jahrzehnten stark an Bedeutung gewonnen hat, ist die der *Frameworks*. Bei Frameworks handelt es sich um Sammlungen von punktuell einsetzbaren Software-Komponenten, auf deren Basis Anwendungssoftware entwickelt werden kann.

Frameworks zeichnen sich dadurch aus, dass sie für sich genommen nicht ausführbar sind, oder jedenfalls nicht auf eine sinnvolle Weise. Vielmehr bestehen Frameworks aus

Schablonen und Hilfsmitteln, die für Entwickler leicht nutzbar sind (oder es jedenfalls sein sollen).

Durch die Verwendung von Frameworks kann Implementationszeit gespart werden, die sonst dafür verwendet werden müsste, immer wieder benötigte, vor allem systemnahe Funktionalität neu zu implementieren. Sie helfen bei der besseren Kapselung von nicht-fachlichem Code, dadurch dass oberhalb des Frameworks idealerweise die fachliche Logik stattfinden sollte, während das Framework selbst die technische Ebene abstrahiert.

Mögliche Nachteile der Verwendung von Frameworks sind eine typischerweise lange Einarbeitungszeit – viele moderne Frameworks sind so umfangreich, dass kaum jemand sie vollständig beherrscht – und die nicht zu unterschätzende Tatsache, dass die Anwendungen an das Framework gebunden sind und damit an die Plattformen, die das Framework unterstützt.

Sommerville untersucht weiterhin verschiedene Arten von Frameworks und stellt typische Entwurfsmuster vor, die beim Einsatz von Frameworks oft Verwendung finden.

2.3. Produktlinien

Bei Produktlinien handelt es sich um eine Reihe mehrerer fachlich verwandter Anwendungen mit einer gemeinsamen technischen Grundlage bzw. gemeinsamen Komponenten.

Sie zeichnen sich dadurch aus, dass sie generische Funktionalität beinhalten, die an verschiedene Kontexte angepasst werden kann. Die verschiedenen Produkte einer Produktlinie sind üblicherweise für verschiedene fachliche Aufgaben konzipiert, teilen sich jedoch einige Funktionalität, die vor allem nicht-fachlichen Charakter haben kann (jedoch nicht ausschließlich).

Ein klassisches Beispiel ist eine *Office*-Produktlinie, die Softwareprodukte für Textverarbeitung, Tabellenkalkulation, Präsentationen und weitere verwandte Aufgaben enthält. Kein einzelnes Produkt kann alles gleichzeitig leisten, aber die Produkte der Produktlinie können sich spezialisieren und sich dabei eine große Menge gemeinsame Funktionalität für Darstellung und Interaktion teilen.

Bei Produktlinien kann die Konfiguration einen hohen Stellenwert haben. Nicht in jedem Fall muss jedes Detail von den Entwicklern festgelegt werden, sondern die Entwickler von Produktlinien entschieden sich häufig für die Implementation einer Komponentenarchitektur, so dass die Produkte sich in hohem Maße anpassen lassen können, zum Beispiel durch Systemadministratoren.

Im Gegensatz zu Frameworks findet bei Produktlinien eine fachliche Konsolidierung statt, nicht ausschließlich eine technische.

Die eng verwandte Architektur der einzelnen Anwendungen gewährleistet, dass die Einarbeitungszeit der Entwickler beim Wechsel zwischen Anwendungen gering bleibt. Außerdem kann Entwicklungs-Infrastruktur (etwa für automatisierte Tests) gemeinsam genutzt werden.

Bei der Gestaltung einer Produktlinie müssen unter Umständen Kompromisse zwischen detaillierten Anforderungen durch die Anwender und der Maximierung der Verwendung der generischen Basis gefunden werden. Mitunter bedeutet eine kleine Veränderung der Anforderungen, dass eine Teilleistung durch die generische Basis nicht mehr

erbracht werden kann und deshalb neu entwickelt werden muss.

Sommerville untersucht in diesem Abschnitt weiterhin verschiedene mögliche Arten der Produktlinien-Spezialisierung, mögliche Variationen der Architektur von Produktlinien sowie den Ablauf der Entwicklung eines neuen Mitglieds für eine Produktlinie.

2.4. Commercial off-the-shelf

Die Abkürzung COTS steht für *„commercial off-the-shelf“* und bezeichnet eine Art von Software, die in einer hochgradig anpassbaren Form entwickelt wurde und dafür ausgelegt ist, von ganz verschiedenen Kunden gekauft und sofort eingesetzt zu werden, ohne dass Veränderungen am Quellcode nötig sind. Oft ist COTS-Software funktional sehr umfangreich.

Die Kosten für COTS-Systeme stehen im Vorhinein sehr genau fest, da die Anschaffung an einen festen Preis gebunden ist und keine Entwicklungskosten anfallen. Erwähnenswert sind allerdings, dass die zu erwartenden Betriebs- und Wartungskosten nicht unterschätzt werden sollten.

Kunden haben zunächst einmal wenig Kontrolle über die angebotene Funktionalität, da der Individualisierung durch die Konfigurationsmöglichkeiten beschränkt ist, die vom Hersteller vorgesehen sind. Alles, was darüber hinausgeht, ist nicht ohne Schwierigkeiten möglich. COTS-Systeme haben eine stetig ungewisse Zukunft in dem Sinne, dass Kunden auf den Hersteller des Systems angewiesen sind. Geht dieser beispielsweise bankrott, ist die Zukunft des Systems ungewiss, da im schlimmsten Fall niemand es weiterentwickeln kann.

In diesem Abschnitt des Kapitels untersucht Sommerville ansonsten vor allem mehrere konkrete Beispiele und unterteilt die COTS-Systeme weiter in zwei Kategorien, die im Folgenden erläutert werden.

2.4.1. COTS-Lösungs-Systeme

Diese Unterkategorie der COTS-Systeme bezeichnet generische, in sich geschlossene Anwendungssysteme für einen bestimmten Einsatzzweck, etwa Universitäts-Verwaltungs-Systeme, Software für Arztpraxen oder ERP-Systeme. Sie bieten alle nötige Funktionalität und sind darauf ausgelegt, mit minimalem Aufwand möglichst schnell eingesetzt werden zu können.

Normalerweise enthalten sie verschiedene fachliche Module für verschiedene Aufgaben, verfügen jedoch über modul-übergreifende Regeln und eine gemeinsame Datenbasis, so dass der Austausch von Informationen zwischen den Modulen auch in stark formalisierter Form problemlos funktioniert.

Trotz des Anspruchs der sofortigen Einsetzbarkeit erfordern sie oft ein hohes Maß an Konfiguration vor der tatsächlichen Einsatztauglichkeit.

2.4.2. Integrierte COTS-Systeme

Als Gegenstück zu den COTS-Lösungs-Systemen bestehen die Exemplare dieser Kategorie aus zwei oder mehr COTS-Produkten, die miteinander verbunden werden. Lösungen

dieser Art erfordern individuellen Entwicklungsaufwand, insbesondere im Bezug auf Wege zum fehlerfreien Datenaustausch.

Ein potenzielles Problem dieser Art von System entsteht durch möglicherweise vorhandene doppelte Funktionalitäten und damit größere Fehlerwahrscheinlichkeiten allein durch die größere Angriffsfläche für mögliche Fehler.

3. Kritik und Ergänzung

Ein Konzept, das in Sommerville's Kapitel zur Software-Wiederverwendung sehr stiefmütterlich behandelt wird und in seiner „Reuse Landscape“ überhaupt nicht auftaucht, ist das des *Open Source*. Dies ist verwunderlich, wenn man bedenkt, wie eng dieses Thema die Software-Wiederverwendung berührt. Deshalb soll in diesem Kapitel als Ergänzung zum Inhalt des *Software Engineering*-Kapitels umrissen werden, wie Open Source funktioniert und welche Rolle dabei die Wiederverwendung von Software spielt.

3.1. Prinzipien von Open Source

Nach gängiger rechtlicher Auffassung [Lemley 1995] handelt es sich bei Software um ein Ergebnis kreativer geistiger Arbeit, am ehesten vergleichbar mit anderen schriftlichen Erzeugnissen aller Art. Somit steht Software in fast allen Ländern der Welt unter dem Schutz des Urheberrechts. Zur Verwendung oder Duplizierung von Software benötigt man deshalb in fast allen Fällen eine explizite Erlaubnis durch den Urheber, die oftmals in Form einer Nutzungslizenz käuflich zu erwerben ist.

Allerdings ist dieses Geschäftsmodell nicht die einzige Möglichkeit, Software sinnvoll einzusetzen. Eine Minderheit unter den Entwicklern von Software stellt ihre Ergebnisse nach dem *Open Source*-Modell der Allgemeinheit zur Verfügung. Die anerkannteste Definition für diesen Begriff ist die der *Open Source Initiative* [Open Source Initiative 2002]. Darin wird für Open Source Software explizit gefordert, dass sie für jedwede Nutzung frei verfügbar sein muss, dass sie von Anwendern angepasst und verändert werden darf und dass diese veränderten Versionen wiederum verteilt werden dürfen. Inhaltlich eng verwandt, jedoch stärker ideologisch behaftet, ist der Begriff der *freien Software* [Free Software Foundation 2012].

Die meisten Firmen, die ihr Geld mit Software verdienen, finanzieren sich durch den Verkauf von Nutzungslizenzen. Das Open-Source-Prinzip zeigt, dass Code nicht zwangsweise ein Geschäftsgeheimnis sein muss. Stattdessen gibt es noch eine ganze Reihe anderer Geschäftsmodelle auf Basis von offenem Code, die im nächsten Abschnitt genauer betrachtet werden.

Ausgehend von der Vorstellung von Software als Geschäftsgeheimnis scheitert die Software-Wiederverwendung in vielen Fällen an den Grenzen von Institutionen. Zwar können Firmen für externe Entwickler APIs und netzwerkbasierte Plattformen anbieten, ohne den Quellcode der Software preisgeben zu müssen, aber der Austausch von Softwarekomponenten auf Code-Basis ist im Allgemeinen nicht möglich. Diese Beschränkung fällt mit Open Source Software weg: Wenn der Code offen liegt, dann kann er ohne Probleme auch unter verschiedenen Firmen ausgetauscht werden. So muss nicht jede

benötigte Softwarekomponente von jeder Firma entwickelt werden, sondern durch allseitigen Austausch können die Produkte aller Beteiligten in kürzerer Zeit weiterentwickelt werden.

Einige bekannte und vielfach eingesetzte Beispiele für Open Source Software sind der Linux-Kernel, der Firefox-Browser von Mozilla oder der Apache-Webserver.

3.2. Open Source und schwarze Zahlen

Was für Geschäftsmodelle sind möglich, wenn der Zugang zur Software nicht in Form von Nutzungslizenzen verkauft wird sondern aufgrund des offenen Quellcodes bereits allgemein offen steht? In folgenden Abschnitt werden einige Möglichkeiten aufgezeigt, auf Basis von Open Source Software Geld zu verdienen. Danach wird dargestellt, welche Vorteile einem Unternehmen durch Engagement im Bereich Open Source entstehen können.

3.2.1. Open-Source-Geschäftsmodelle

Eine offensichtliche Möglichkeit ist der **Verkauf von Services und Support rund um die Software** statt des Verkaufs von Nutzungsrechten an der Software selbst. Bekanntermaßen ist die einmalige Anschaffung von Software selten der größte Kostenfaktor, hinzu kommen Unterstützung im Fehlerfall, Dokumentation oder Schulungen. All diese Dinge können prinzipiell von den Herstellern der Software geleistet werden und bieten dabei natürlich ein beträchtliches Gewinnpotenzial.

Eine Variation dieser Herangehensweise ist das **„Commoditize complements"-Prinzip**. Dabei werden neben Open-Source-Komponenten auch nutzungbeschränkte Komponenten angeboten, die üblicherweise erst relevant werden, wenn die Open-Source-Teile in großem Umfang genutzt werden. Beispielsweise könnte ein Hersteller von Datenbanksystemen eine Basisversion seines Produkts als Open Source veröffentlichen, so dass Hobbyentwickler, Angestellte und kleine Firmen ohne Einschränkungen damit arbeiten können und neue Produkte damit entwickeln können. Für eine erweiterte Version mit Expertenfeatures (im Beispiel der Datenbanksysteme zum Beispiel die Fähigkeit zur Datensicherung im laufenden Betrieb, Verschlüsselung oder Replizierung auf mehreren Servern) kann es dann Zugangsbeschränkungen in Form von Lizenzkosten geben.

Wie bereits im vorigen Abschnitt erläutert kann bereits die **Kostenteilung bzw. -vermeidung** Motivation genug für Open Source sein. Wenn Firmen Software entwickeln, die außerhalb ihrer Kernkompetenzen liegt und eher infrastrukturellen Charakter hat, dann kann es sich schnell lohnen, diese Software nicht allein zu entwickeln sondern gemeinschaftlich mit anderen interessierten Firmen und Personen in Form eines Open-Source-Projekts. Zum Beispiel betreiben sehr viele Firmen eigene Webseiten. Anstatt dass nun jede Firma einen eigenen Webserver programmiert oder einkauft, kann auch ein gemeinschaftlicher Webserver mit allen benötigten Features als Open-Source-Projekt entwickelt werden. Auf diese Weise bekommen alle Beteiligten schneller, was sie brauchen um am besten alle verfügbaren Ressourcen auf ihre Kernkompetenzen zu konzentrieren.

Nicht zuletzt bietet auch das Modell **Spenden und Crowdfunding** erwiesenermaßen

[Hemmerich 2012] das Potenzial, Projekte zu finanzieren, an denen dem interessierten Kundenkreis etwas liegt. Natürlich ist diese Art der Finanzierung nicht weniger risikobehaftet als andere, sondern im Gegenteil oftmals von Faktoren abhängen, die außerhalb der Kontrolle des Unternehmens liegen.

3.2.2. Geschäftsvorteile durch Open-Source-Engagement

Die Frage, ob ein Unternehmen sich im Open-Source-Bereich einbringen sollte, läuft auf die Höhe des ökonomischen Nutzens eines solchen Engagements zurück. Auf den ersten Blick erscheint es absurd, sich von der freien Weitergabe von Arbeitsergebnissen einen Geschäftsvorteil zu erhoffen. Doch egal ob ein eigenes Produkt offengelegt oder Arbeiten an einem bestehenden Open-Source-Projekt veröffentlicht werden sollen, mögliche Geschäftsvorteile existieren in beiden Fällen.

Bei Open-Source-Veröffentlichung eigener Produkte:
Ein möglicher Geschäftsvorteil, der sich durch die Open-Source-Veröffentlichung eines Produktes ergeben kann, ist die Minderung von Hemmschwellen bei Kunden. Auf diese Weise wird die Erschließung des Marktes vereinfacht, indem der Einstieg ins Produkt erleichtert wird, so dass Verpflichtungen und Kaufverträge wegfallen. Dies kann Hand in Hand mit dem im vorigen Abschnitt beschriebenen „Commoditize complements"-Geschäftsmodell gehen, muss es aber nicht. Der Hersteller hat die Möglichkeit, durch Softwarequalität zu beeindrucken, ohne dafür irgendwelche Werbekosten aufbringen zu müssen.

Weiterhin wird die Nutzung von externer Kompetenz für die Entwicklung und die Qualitätssicherung erleichtert. Viele Programmierer arbeiten in ihrer Freizeit an Open-Source-Projekten mit, die sie für wichtig halten oder selbst nutzen. Für eine Firma bedeutet dies kurzfristig gesehen kostenlos erbrachte Arbeit an ihrem Produkt, mittel- und langfristig allerdings auch mögliche Verkürzungen von Training und Recruiting, wenn zukünftige Arbeitnehmer bereits vor ihrer Anstellung an der Software arbeiten können.

Andersherum wird durch Open-Source-Engagement eine Firma auch potenziell interessanter für externe Entwickler. Bei der Auswahl des Arbeitgebers spielen bekanntlich unzählige Faktoren eine Rolle, aber das Bewusstsein, an Open Source Software entwickeln zu können, ist für viele Entwickler ein durchaus positiver Faktor.

Nicht zuletzt wirkt eine Open-Source-Veröffentlichung eines Softwareprojektes auch der „Versteinerung" der Entwicklerbasis entgegen – soll heißen, dass das Entwicklerteam in stetigem Kontakt und Austausch von Ideen und Feedback mit der Außenwelt steht, statt sich ausschließlich um sich selbst zu drehen. Dem Problem, dass Entwickler manchmal den Bezug zum Kontext verlieren und sich in Details verrennen, wird auf diese Weise entgegengewirkt.

Bei Beteiligung an bestehenden Open-Source-Projekten:
Viele Firmen setzen existierende Open Source Software intern ein, zum Beispiel als Content Management System oder zur Systemverwaltung. Für die Verwendung von Open Source Software in Fällen, in denen auch eine proprietäre Alternative zur Verfügung

stünde, spricht zum Beispiel die uneingeschränkte Verfügbarkeit des Quellcodes. Wenn die Software in irgendeiner Hinsicht nicht den Anforderungen entspricht, ist man nicht auf den guten Willen des Herstellers angewiesen, sondern kann nötige Veränderungen einfach selbst vornehmen. Hinzu kommt, dass die Verfügbarkeit des Quellcodes eine große Hilfe bei der Fehlerdiagnose und -behebung darstellt.

Die meisten Open-Source-Projekte werden nach dem sogenannten *Meritocracy*-Prinzip [Apache Software Foundation 2012] organisiert. Das bedeutet, dass die Ziele und die Entwicklungsrichtung des Projekts durch diejenigen bestimmt wird, die am meisten zum Projekt beitragen. Dies kann eine große Hilfe für Firmen sein, die Open Source Software intern einsetzen und anpassen: Wenn diese ihre Veränderungen wieder mit dem Projekt teilen, erhalten sie auf diese Weise Einfluss bei der Ausgestaltung des Projekts und können Kompatibilitätsprobleme vermeiden und das Projekt in eine für die Firma vorteilhafte Richtung lenken.

Ebenso führen intern gepflegte Veränderungen von Open Source Software zu organisatorischen Schwierigkeiten, wenn neue Versionen der Software erscheinen und die Veränderungen jedes Mal wieder neu eingepflegt werden müssen. Dieser Aufwand ist komplett vermeidbar, wenn die Veränderungen direkt im Open-Source-Projekt entwickelt und gepflegt werden. Auf diese Weise kann viel Entwicklungszeit gespart werden.

4. Fallbeispiel: Steve Yegge's Google-Kritik

Ein praxisnahes und anschauliches Beispiel für die Auswirkungen von fehlender Weitsicht in Bezug auf Software-Wiederverwendung lieferte Steve Yegge im Oktober 2011. Yegge verfasste einen Text [Yegge 2011] über die Firmen- und Softwareentwicklungs-Politik seines Arbeitgebers, Google Inc. Dieser Text war eigentlich nur für seine Arbeitskollegen vorgesehen, doch Yegge stellte bei der Veröffentlichung im Google+-Netzwerk versehentlich ein, dass der Text weltöffentlich sichtbar sein sollte. Binnen kurzer Zeit stieß der Text auf so viel Interesse, dass diverse Kopien existierten, bevor Yegge ihn wieder aus dem öffentlichen Netz entfernen konnte. Er entschloss sich daraufhin, die Verbreitung des Textes nicht zu unterbinden. Sergey Brin, Vorstandsmitglied bei Google Inc., gab im Nachhinein an, dass Yegge's Anstellung durch dieses Versehen nicht in Gefahr sei.

Inhaltlich geht es in dem Text unter anderem um die von Yegge beobachteten Schwächen von Google hinsichtlich der Wiederverwendung von Software-Komponenten und dem Anbieten von Schnittstellen nach außen und für andere Teams. Yegge zieht Vergleiche zu seinem früheren Arbeitgeber Amazon und kritisiert Googles Umgang mit dem Bedarf an Interoperabilität zwischen ihren einzelnen Produkten.

Im Folgenden werden einige Passagen aus dem Text zitiert, die hinsichtlich Software-Wiederverwendung von Relevanz sind.

> 1) All teams will henceforth expose their data and functionality through service interfaces.
> 2) Teams must communicate with each other through these interfaces.
> 3) (...) The only communication allowed is via service interface calls over the network.

> 4) It doesn't matter what technology they use. (. . .)
> 5) (. . .) [T]he team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
> 6) Anyone who doesn't do this will be fired.

Diese Darstellung in Yegge's Worten gibt eine Maßnahme wieder, die Jeff Bezos ungefähr im Jahr 2002 bei Amazon eingeleitet hat. In diesem Zuge wurde dort flächendeckend das Prinzip eingeführt, dass sämtliche in-house entwickelte Software uneingeschränkt auf Service-Basis verfügbar sein muss. Das war laut Yegge der entscheidende Faktor, der es Amazon ermöglichte, blitzschnell auf den sich verändernden Markt zu reagieren und technische Ressourcen vom Warenverkauf (speziell Bücher), der sich in hohem Maße reduzierte, zum Cloud Computing zu verlagern. Produkte wie Amazon S3 oder Amazon AWS konnten extrem schnell entwickelt und öffentlich gemacht werden, dadurch dass beinahe alle nötigen Bestandteile bereits in marktreifer Form vorhanden waren.

Diese Mentalität, die Yegge als Ausprägung von Jeff Bezos' außergewöhnlicher Weitsicht ansieht, vermisst er bei Google. So sei es angeblich sehr schwer, ein Entwicklerteam bei Google dazu zu bringen, für ihre Software wohldefinierte programmatische Schnittstellen anzubieten – dies gehöre dort nicht zur Firmenkultur und sei der Grund für die oftmals unzufriedenstellende Interoperabilität zwischen Googles einzelnen Produkten:

> That one last thing that Google doesn't do well is Platforms. We don't understand platforms. We don't "get" platforms. (. . .) It's a big stretch even to get most teams to offer a stubby service to get programmatic access to their data and computations. Most of them think they're building products.

Dabei unterstreicht Yegge die Wichtigkeit der Anpassbarkeit der Software insbesondere in dem Sinne, dass es sich lohne, Plattformen anzubieten, auf deren Basis andere Entwickler weitere Produkte entwickeln können. Denn, so Yegge, der Versuch ist sinnlos, das perfekte Produkt für alle entwickeln zu wollen:

> But when we take the stance that we know how to design the perfect product for everyone, and believe you me, I hear that a lot, then we're being fools. You can attribute it to arrogance, or naivete, or whatever – it doesn't matter in the end, because it's foolishness. There IS no perfect product for everyone.

Yegge führt weiter aus, dass eine hohe Anpassbarkeit ein Garant für den Erfolg eines Softwareproduktes sein kann, indem er das seinerzeit tagesaktuelle Google+ mit dem etablierten Konkurrenten Facebook vergleicht und Defizite von Google+ hinsichtlich seiner Plattform-Fähigkeiten herausarbeitet.

Jedoch, so Yegge, ist Google+ in dieser Hinsicht keine Ausnahme, sondern folgt einem wiederkehrenden Muster, dass Google-Produkte eine unzureichende API haben, was externe Entwickler davon abhält, sie weiter auszubauen.

> Google+ is a prime example of our complete failure to understand platforms (...) The Google+ platform is a pathetic afterthought. We had no API at all at launch, and last I checked, we had one measly API call.

Er diskutiert weiter ausführlich die Unterschiede zur Konkurrenz Facebook, wo sich die Plattform-Mentalität augenscheinlich ebenfalls durchgesetzt habe. Den Erfolg von Facebook schreibt Yegge nur an zweiter Stelle dem eigentlichen Facebook-Kernprodukt zu, aber an erster Stelle der Erweiterbarkeit und der Vielfalt an zu Verfügung stehenden Anwendungen für die Facebook-Plattform.

> Google+ is a knee-jerk reaction, a study in short-term thinking, predicated on the incorrect notion that Facebook is successful because they built a great product. But that's not why they are successful. Facebook is successful because they built an entire constellation of products by allowing other people to do the work. So Facebook is different for everyone.

Natürlich sind dies nicht die einzigen Themen, die Yegge in seinem Text anschneidet, und umgekehrt deckt er konzeptuell auch keinen allzu großen Teil von Sommerville's Reuse Landscape ab. Dennch zeigen gerade diese Stellen – Zitate aus dem Munde eines angesehenen Softwareentwicklers mit sehr viel Erfahrung – wie wichtig die richtige Antizipation von Software-Wiederverwendung für den Erfolg eines Produktes oder einer ganzen Firma sein kann.

5. Fazit

Software-Wiederverwendung ist ein ebenso breit gefächertes wie wichtiges Thema. Die von Ian Sommerville vorgebrachten Versuche zur Strukturierung leisten einen wertvollen Überblick über das Feld, sind jedoch nicht als erschöpfend zu werten, da zumindest der Aspekt der Open Source Software dort gänzlich untergeht, obwohl er in der modernen Software-Welt einen großen Beitrag zur Wiederverwendung leistet. Das von Steve Yegge dargestellte Beispiel zu Services und Plattformen verfolgt wiederum einen ganz anderen Ansatz, konnte dabei allerdings ebenfalls unmittelbar die Praxisrelevanz verdeutlichen.

Techniken zur Software-Wiederverwendung werden immer sinnvoller und relevanter, da sich der gesamte Pool der Software auf der Welt immer weiter mehrt und es angesichts der Komplexität moderner Software nicht zeitgemäß erscheint, jedes Problem wieder und wieder zu lösen. Dabei sollte man jedoch nicht die Fallstricke aus den Augen verlieren, derer es sicher große Mengen gibt.

Literatur

Apache Software Foundation 2012
Apache Software Foundation: *How the ASF works*. https://www.apache.org/foundation/how-it-works.html#meritocracy. Version: 2012, Abruf: 16. März 2012

Free Software Foundation 2012
Free Software Foundation: *What is free software?* https://www.gnu.org/philosophy/free-sw.html. Version: Februar 2012, Abruf: 16. März 2012

Hemmerich 2012
Hemmerich, Lisa: *Crowdfunding: Double Fine Productions erhält über 3,3 Millionen US-Dollar*. http://www.netzwelt.de/news/91350. Version: 14. März 2012, Abruf: 16. März 2012

Lemley 1995
Lemley, Mark A.: *Convergence in the Law of Software Copyright?* In: *Berkeley Technology Law Journal* 10 (1995), Nr. 1

Open Source Initiative 2002
Open Source Initiative: *The Open Source Definition (Annotated)*. http://www.opensource.org/osd.html. Version: 2002, Abruf: 16. März 2012

Sommerville 2010
Sommerville, Ian: *Software Engineering*. 9th edition. Pearson, 2010 (International Computer Science Series). – ISBN 9780137053469

Yegge 2011
Yegge, Steve: *Google Platforms Rant*. Öffentlich durch Genehmigung des Autors. https://plus.google.com/112678702228711889851/posts/eVeouesvaVX. Version: 12. Oktober 2011, Abruf: 16. März 2012