Linux
Professional
Institute

Open Source Essentials

Version 1.0
Deutsch

050

Table of Contents

Linux Professional Institute

Thema 051: Grundlagen von Software

051.1 Softwarekomponenten

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 051.1

Gewichtung

2

Hauptwissensgebiete

- Verständnis des Konzepts von Quellcode und Codeausführung
- Verständnis des Konzepts von Compilern und Interpretern
- Verständnis des Konzepts von Softwarebibliotheken

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Quellcode
- Ausführbare Programme
- Bytecode
- Maschinencode
- Compiler
- Linker
- Interpreter
- Virtuelle Maschine zur Laufzeit
- Algorithmus
- Softwarebibliotheken
- Statisches und dynamisches Linking

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	051 Grundlagen von Software
Lernziel:	051.1 Softwarekomponenten
Lektion:	1 von 1

Einführung

Freie und *Open Source Software*, oft als FOSS abgekürzt, spielt in unserem Alltag eine wichtige Rolle, ohne dass wir uns dessen bewusst sind. FOSS verbirgt sich zum Beispiel hinter all unseren Aktivitäten im Internet: auf dem Computer, auf dem wir Webseiten im Browser betrachten, oder auf Servern, die diese Webseiten speichern und ausliefern, sobald wir sie aufrufen.

Was ist Software?

Bevor wir auf die Besonderheiten von freier und quelloffener Software eingehen, müssen wir klären, was *Software* eigentlich ist. Beginnen wir mit einer sehr allgemeinen Beschreibung: Software ist der nicht-physische, immaterielle Teil von Computern in jeder Form. Software sorgt dafür, dass die physischen Teile (die *Hardware*) des Computers zusammenwirken und dass der Computer Befehle annehmen und Aufgaben ausführen kann.

Ein Smartphone, ein Laptop oder ein Server im Rechenzentrum ist im ausgeschalteten Zustand nur eine Maschine aus Metall und Kunststoff. Wird sie eingeschaltet, startet Software in Form kodierter Befehlssequenzen, die die Komponenten dieser Maschine steuern. Das macht es dem Benutzer möglich, mit der Maschine zu interagieren und durch den Aufruf einzelner

Anwendungen bestimmte Aufgaben erfüllen.

Es ist die Arbeit der *Softwareentwickler*, die Aufgaben zu analysieren, die der Computer erfüllen soll, und sie so zu spezifizieren, dass der Computer sie ausführen kann. Die von Entwicklern verwendeten Werkzeuge sind so zahlreich und vielfältig wie die Aufgaben, die die Software erfüllt.

Einige Aufgaben von Software sind eng mit der Hardware und der Architektur des Rechners verbunden, beispielsweise die Adressierung und Verwaltung des Speichers oder die Handhabung verschiedener Prozesse. *Systemprogrammierer* arbeiten daher nahe an der Hardware.

Anwendungsentwickler hingegen haben mehr die Benutzer im Blick und programmieren Anwendungen, mit denen diese ihre Aufgaben sowohl effizient als auch intuitiv erledigen. Ein Beispiel für eine komplexe Anwendung ist ein Textverarbeitungsprogramm, das alle Funktionen zur Textformatierung in Menüs oder Schaltflächen bereitstellt und den Text auch so anzeigt, wie er zuletzt gedruckt werden könnte.

Allgemein ist ein *Algorithmus* ein definierter Weg, ein Problem zu lösen. Um beispielsweise einen Durchschnitt zu berechnen, besteht der übliche Algorithmus darin, mehrere Werte zu addieren und die Summe durch die Gesamtzahl der Werte zu dividieren. Traditionell entwarfen Programmierer die notwendigen Algorithmen — heute werden sie zunehmend auch von künstlicher Intelligenz erzeugt.

Die in dieser Lektion vorgestellten Konzepte helfen, Stärken und Risiken von FOSS zu verstehen, fundierte Entscheidungen zu treffen und sogar einzuschätzen, ob Sie Software entwickeln wollen.

Programmiersprachen

Programmiersprachen sind hochgradig strukturierte, künstliche Sprachen, in denen dem Computer gesagt wird, was er zu tun hat. Programme werden in der Regel in Textform geschrieben, aber einige Sprachen auch in grafischer Form. Softwareentwickler schreiben Anweisungen (den *Code*) für den Computer in einer solchen künstlichen Sprache. Die Computerhardware kann diesen Code jedoch nicht unmittelbar, sondern nur als eine Reihe von Bitmustern ausführen, die sich im Speicher befinden und *Maschinencode* oder *Maschinensprache* genannt werden. Alle Programmiersprachen werden entweder von einem *Compiler* in Maschinencode umgewandelt oder von einem anderen Maschinencodeprogramm, dem *Interpreter*, ausgewertet, damit die Hardware die Anweisungen ausführen kann.

Zu den derzeit am weitesten verbreiteten Programmiersprachen gehören Python, JavaScript, C, C++, Java, C#, Swift und PHP. Jede dieser Programmiersprachen hat ihre Stärken und Schwächen, und die Wahl der Sprache hängt vom Projekt und den Bedürfnissen der Entwickler ab. So ist Java

beispielsweise eine beliebte Wahl für die Entwicklung großer Unternehmensanwendungen, während Python häufig bei wissenschaftlichen Berechnungen und Datenanalysen zum Einsatz kommt.

Entwickler haben bei der Gestaltung von Programmiersprachen beeindruckende Kreativität an den Tag gelegt. Ursprünglich gab es nur *low-level* (hardwarenahe) Sprachen, die den Anweisungen im Computer ähnelten. Sie wurden immer mehr zu *höheren Programmiersprachen*, die versuchen, komplexe Kombinationen von Anweisungen in kurzen Sequenzen abzubilden. Einige Sprachen spiegeln die Art und Weise wider, wie Menschen natürlicherweise denken, und bewahren gleichzeitig die für eine korrekte Ausführung erforderliche Stringenz.

Gegenwärtig sind etwa 400 Programmiersprachen bekannt, von denen viele jedoch nur in Nischenanwendungen oder Legacy-Umgebungen zum Einsatz kommen. Jede wurde entwickelt, um bestimmte Aufgaben zu lösen.

Merkmale, Syntax und Struktur von Programmiersprachen

Die Wahl der Programmiersprache hat erheblichen Einfluss auf Leistung, Skalierbarkeit und Entwicklung eines Softwareprojekts. In den folgenden Abschnitten werden wichtige Sprachelemente erläutert.

Merkmale von Programmiersprachen

Zu den allgemeinen Merkmalen und Eigenschaften von Programmiersprachen gehören:

Gleichzeitigkeit (Concurrency)

Concurrency bezeichnet die zeitgleiche Bearbeitung mehrerer Aufgaben, entweder durch die Ausführung auf verschiedenen Hardware-Prozessoren oder durch die abwechselnde Bearbeitung verschiedener Aufgaben durch einen einzelnen Prozessor. Der Grad der Gleichzeitigkeit, den eine Programmiersprache unterstützt, kann sich erheblich auf Leistung und Skalierbarkeit auswirken, insbesondere bei Anwendungen, die Echtzeitverarbeitung oder große Datenmengen erfordern. Ein einzelner Arbeitsschritt wird als *Prozess*, *Aufgabe* oder *Thread* bezeichnet.

Speicherverwaltung (Memory Management)

Unter Speicherverwaltung versteht man die Zuweisung und Freigabe von Speicher in einem Programm. Je nach Sprache oder Laufzeitumgebung erfolgt die Speicherverwaltung manuell durch den Programmierer oder automatisch. Eine ordnungsgemäße Speicherverwaltung entscheidet darüber, ob ein Programm den Speicher effizient nutzt oder ihm Speicher fehlt, wodurch weitere Probleme entstehen. Ist ein Programm nicht in der Lage, ungenutzten Speicher freizugeben, entsteht ein *Speicherleck* (*Memory Leak*), durch das der

Speicherverbrauch allmählich ansteigt, bis die Leistung des Programms nachlässt oder es abstürzt.

Gemeinsam genutzter Speicher (Shared Memory)

Shared Memory ist eine Art Kommunikationsmechanismus zwischen mehreren Prozessen, über den diese einen gemeinsamen Speicherbereich lesen und bearbeiten. Gemeinsam genutzter Speicher ist in Hardware beispielsweise in Laufwerken üblich und bietet auch eine effiziente Möglichkeit, Daten zwischen Prozessen auszutauschen. Der Mechanismus erfordert jedoch sorgfältige Synchronisierung und Verwaltung, um eine Beschädigung der Daten zu verhindern. Ein Fehler, der als *Race Condition* (*Wettlaufsituation*) bekannt ist, tritt auf, wenn ein Prozess eine unvorhergesehene Änderung an Daten vornimmt, während ein anderer Prozess diese nutzt.

Nachrichtenaustausch (Message Passing)

Message Passing ist ein Kommunikationsmechanismus zwischen Prozessen, über den diese Daten austauschen und Aktivitäten koordinieren. Er wird häufig in der nebenläufigen (concurrent) Programmierung für die Kommunikation zwischen Prozessen genutzt und durch Mechanismen wie Sockets, Pipes oder Warteschlangen (Message Queues) implementiert.

Automatische Speicherbereinigung (Garbage Collection)

Garbage Collection ist eine automatische Speicherbereinigung, die einige Programmiersprachen nutzen, um Speicher zurückzugewinnen, der nicht mehr verwendet wird, während ein Prozess läuft. Sie kann dazu beitragen, Speicherlecks zu verhindern, und es Entwicklern erleichtern, korrekten und effizienten Code zu schreiben. Sie kann aber auch Overhead in der Performance verursachen und die Kontrolle über das exakte Verhalten des Programms erschweren.

Datentypen (Data Types)

Datentypen bestimmen, welche Art von Informationen im Programm darstellbar sind. Datentypen können in der Sprache vordefiniert oder benutzerdefiniert sein und umfassen Ganzzahlen, Gleitkommazahlen (d.h. Näherungswerte für reelle Zahlen), Zeichenketten, Arrays und andere.

Eingabe/Ausgabe, E/A (Input/Output, I/O)

Eingabe und Ausgabe sind Mechanismen zum Lesen und Schreiben von Daten in ein und aus einem Programm. Die Eingabe kann aus einer Vielzahl von Quellen stammen, wie etwa Benutzerklicks und Tastatureingaben, einer Datei oder einer Netzwerkverbindung, während die Ausgabe an eine Vielzahl von Zielen gesendet werden kann, wie beispielsweise eine Anzeige, eine Datei oder eine Netzwerkverbindung. E/A erlaubt Programmen die Interaktion mit der Außenwelt und den Austausch von Informationen mit anderen Systemen.

Fehlerbehandlung (Error Handling)

Error Handling erkennt Fehler, die während der Ausführung eines Programms auftreten, und reagiert darauf. Dazu gehören Fehler wie die Division durch Null oder eine angeforderte Datei, die nicht gefunden wird. Durch das Error Handling kann ein Programm auch bei auftretenden Fehlern weiterlaufen, was dessen Zuverlässigkeit erhöht.

Die genannten Konzepte sind von grundlegender Bedeutung, um zu verstehen, wie Programmiersprachen funktionieren und wie man effizienten und wartbaren Code schreibt.

Syntax von Programmiersprachen

Die *Syntax* einer Programmiersprache beschreibt die Regeln für das Schreiben von Programmanweisungen und -ausdrücken. Es ist wichtig, dass die Syntax *wohldefiniert* und *konsistent* ist, damit Programmierer ihren Code überhaupt schreiben und verstehen können. Die folgenden sind Bausteine der meisten Programmiersprachen:

Prozeduren und Funktionen

Prozeduren und Funktionen dienen dazu, wiederverwendbare Codeblöcke zu definieren, die mehrfach aufgerufen werden.

Variablen

Variablen bilden Teile des Speichers ab und enthalten Daten, die verändert und zwischen Prozeduren und Funktionen weitergegeben werden können.

Operatoren

Operatoren sind Schlüsselwörter oder Symbole (wie `+` und `-`), die Variablen *Werte* zuweisen und arithmetische Operationen durchführen.

Kontrollstrukturen

Im allgemeinen wird Programmcode in der Reihenfolge ausgeführt, in der er geschrieben wurde; *bedingte Anweisungen* ändern jedoch den Ablauf der Ausführung. Welcher Code als nächstes ausgeführt wird, hängt von verschiedenen Bedingungen ab: dem Inhalt des Speichers, der Tastatureingabe, den aus dem Netzwerk ankommenden Paketen usw. Die *Schleifenanweisung* (*Loop Statement*), eine spezielle Form der bedingten Anweisung, dient dazu, dieselben Operationen an einer Reihe von Datensätzen auszuführen. Eine *Ausnahme* (*Exception*), die bei Auftreten eines Fehlers speziellen Code aufruft, ist eine weitere Kontrollstruktur.

Syntax und Verhalten dieser Konstrukte können von Programmiersprache zu Programmiersprache unterschiedlich sein, und die Wahl der Sprache hat großen Einfluss auf Lesbarkeit und Wartbarkeit des Codes.

Bibliotheken

Eine gute Programmiersprache sollte die Entwicklung von Programmen und die Wiederverwendung von bestehendem Code erleichtern. Viele Programmiersprachen verfügen über einen Mechanismus, mit dem sich Prozeduren und Funktionen in Teile gliedern lassen, die in anderen Programmen wiederverwendet werden können.

Eine *Bibliothek* (*Library*) ist eine Sammlung von Prozeduren und Funktionen zur Unterstützung eines bestimmten Features oder Ziels, zusammengefasst in einer einzigen Datei. Die Verfügbarkeit vieler einfach einzusetzender Bibliotheken ist eine wichtige Anforderung an eine gute Programmiersprache. Python zum Beispiel gilt weithin als eine gute Sprache für die Entwicklung KI-bezogener Programme, weil es eine Reihe von Bibliotheken für die KI-Verarbeitung gibt.

Mit der zunehmenden Größe und Komplexität von Programmen werden Bibliotheken als fertige Bausteine immer wichtiger. Dies gilt insbesondere für die Open-Source-Welt, wo Code, den andere erstellt haben, gerne übernommen und wiederverwendet wird. Infolgedessen hat sich für jede Programmiersprache ein Ökosystem von Bibliotheken entwickelt, und Paketmanager wie `composer` für PHP, `pip` für Python und `gems` für Ruby machen die Installation von Bibliotheken einfach.

Bibliotheken sind auch in kompilierten Sprachen wichtig. Das Kombinieren mehrerer Binärdateien und vorkompilierter Bibliotheken in einer einzigen ausführbaren Datei bezeichnet man als *Linken*; das Werkzeug, das diesen Vorgang ausführt, ist ein *Linker*. Es gibt zwei Arten des Linkens: Beim *statischen Linken* wird nur der notwendige Code einer Bibliothek in die ausführbare Datei der Anwendung aufgenommen; beim *dynamischen Linken* wird eine im System installierte Bibliothek von allen Anwendungen, die diese Bibliothek verwenden, gemeinsam genutzt. Heute ist dynamisches Linken der bevorzugte Ansatz, weil die ausführbaren Anwendungsdateien kleiner und der Speicherverbrauch zur Laufzeit geringer ist.

Da mehrere Programme dieselben Bibliotheken nutzen, können Unterschiede zwischen den Versionen einer Bibliothek ein Problem darstellen. Schauen wir uns daher kurz an, wie Versionsnummern aufgebaut sind: Üblich ist die *semantische Versionierung*, die Versionen durch drei durch Punkte getrennte Zahlen bezeichnet. Eine typische Versionsangabe wäre beispielsweise `2.39.16` mit der Hauptversion 2 (die sich nur selten, vielleicht jährlich, ändert), der Nebenversion 39 innerhalb der Hauptversion (die mehrfach pro Jahr aktualisiert werden kann, um wichtige Funktionsänderungen anzuzeigen) und der schnelllebigen Revision 16 (die sich schon aufgrund einer einzigen Fehlerbehebung ändert). Spätere Versionen und Revisionen haben höhere Zahlen.

Ein sehr einfaches Beispiel

Schauen wir uns ein *sehr* einfaches Beispiel für ein Computerprogramm in der Sprache Python an, um eine ungefähre Vorstellung von einigen der genannten Strukturelemente zu bekommen.

In natürlicher Sprache ausgedrückt, soll das Programm Folgendes tun: “Fordere den Benutzer auf, eine Zahl einzugeben, und prüfe, ob diese Zahl gerade oder ungerade ist. Gib zuletzt das Ergebnis der Prüfung aus.”

Und hier ist der Code, den wir in der Datei `simpleprogram.py` speichern können:

```
num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("The given number is EVEN.")
else:
   print("The given number is ODD.")
```

Selbst in diesen wenigen Codezeilen finden wir viele der oben genannten Merkmale und Syntaxelemente:

1. In Zeile 1 setzen wir die *Variable* `num` und weisen ihr einen *Wert* mit dem *Operator* `=` zu.
2. Der zugewiesene Wert entspricht der *Eingabe* des Benutzers (über die *Funktion* `input()`). Außerdem sorgt die Funktion `int()` dafür, dass diese Eingabe in den *Datentyp* Ganzzahl umgewandelt wird, sofern dies möglich ist. Der Ausdruck, der innerhalb von Klammern an eine Funktion übergeben wird, heißt *Parameter* oder *Argument*.
3. Gibt der Benutzer eine Zeichenkette aus Buchstaben ein, würde Python einen Fehler als Teil des *Error Handling* ausgeben. Wird eine Dezimalzahl eingegeben, wandelt die Funktion `int()` sie in die Basiszahl um, beispielsweise `5.73` in `5`.
4. In den folgenden Zeilen steuert die *Kontrollstruktur*, die eine *Bedingung* angibt, mit den Schlüsselwörtern `if` und `else`, was in jedem der beiden möglichen Fälle geschieht (die Zahl ist entweder gerade oder ungerade).
5. Zunächst testet der Modulo-Operator `%`, ob (`if`) die Division der eingegebenen Zahl durch 2 den Wert 0 ergibt (d.h. kein Rest) — in diesem Fall ist die Zahl gerade. Das doppelte `==` ist der Vergleichsoperator “ist gleich”, der sich vom Zuweisungsoperator `=` in Zeile 1 unterscheidet.
6. Im anderen Fall (`else`), d.h. wenn die Division durch 2 ein Ergebnis ungleich 0 ergibt, muss die eingegebene Zahl ungerade sein.
7. In beiden Fällen gibt die Funktion `print()` das Ergebnis als *Ausgabe* in Textform zurück.

Und so sieht es aus, wenn wir das Programm in der Befehlszeile ausführen:

```
$ python simpleprogram.py
Enter a number: 5
The given number is ODD.
```

Wenn man bedenkt, wie viel Sprachlogik bereits in diesem kleinen Beispiel steckt, bekommt man eine Vorstellung davon, wozu komplexe Software, über Tausende von Dateien verteilt, in der Lage ist — zum Beispiel *Betriebssysteme* wie Microsoft Windows, macOS oder Linux, die die gesamte Hardware eines Computers verfügbar machen und gleichzeitig dafür sorgen, dass die Benutzer alle gewünschten Anwendungen installieren und für die Arbeit oder zum Spaß nutzen können.

Maschinencode, Assemblersprache und Assembler

Wie bereits erwähnt, kann Hardware nur eine Reihe von Bitmustern, den so genannten Maschinencode, direkt ausführen. Die Central Processing Unit (CPU), also der Prozessor, liest ein Bitmuster in Einheiten von einem Wort (8 bis 64 Bits) aus dem Speicher und führt die entsprechende Anweisung aus. Einzelne Anweisungen sind recht einfach, beispielsweise “Kopiere den Inhalt von Speicherplatz A nach Speicherplatz B”, “Multipliziere den Inhalt von Speicherplatz C mit dem Inhalt von Speicherplatz D” oder “Lies die Daten, die an Adresse X im Gerät angekommen sind”. In der Ära der 8-Bit-CPUs konnten sich manche Programmierer noch alle im Maschinencode verwendeten Bitmuster merken und direkt Programme schreiben. Heutzutage hat sich die Zahl der Anweisungsmuster vervielfacht, und sich alle diese Muster zu merken, wäre unpraktikabel.

Maschinencode ist eine Abfolge von Bitmustern (0 und 1), was für den Menschen nicht intuitiv ist. Um das Programmieren intuitiver zu machen, wurde die *Assemblersprache* entwickelt, in der die Anweisungen Namen erhalten und durch Zeichenketten spezifiziert sind. In Assemblersprache werden Anweisungen, die exakt dem Maschinencode entsprechen, nacheinander geschrieben und könnten so aussehen:

```
move    [B], [A]       Kopiere den Inhalt von Speicher A nach Speicher B
multi   R1, [C], [D]   Multipliziere den Inhalt von Speicher C mit dem Inhalt
                       von Speicher D
input   R1, [X]        Lies die Daten, die im Gerät an der Adresse X angekommen sind
```

Eine Anweisung in Assemblersprache entspricht einer Anweisung in Maschinencode, die die Hardware verstehen und ausführen kann. Zu den Vorteilen der Assemblersprache gegenüber Maschinensprache gehören:

Bessere Lesbarkeit und Wartbarkeit

Assemblersprache ist viel einfacher zu lesen und zu schreiben als Maschinencode, was es Programmierern erleichtert, Code zu verstehen, zu debuggen und zu warten.

Automatisierung der Adressberechnung

Die Programmierung von Maschinencode kann zwar auch das Konzept von Variablen und Funktionen nutzen, aber alles muss in Form von *Speicheradressen* ausgedrückt werden. Assemblersprache ordnet den Speicheradressen auch Namen zu, was es einfacher macht, die Logik eines Programms auszudrücken.

Da Assemblersprache Zugriff auf alle Funktionen der Hardware hat, wird sie häufig in den folgenden Situationen verwendet:

Architekturabhängiger Teil des Betriebssystems

Spezielle Anweisungen, die in einer CPU-Architektur für den Zugriff auf Initialisierungs- und Sicherheitsfunktionen spezifisch sind, können nur in Assemblersprache erfolgen.

Entwicklung von Low-Level-Systemkomponenten

Assemblersprache kommt bei der Entwicklung von Systemkomponenten zum Einsatz, die direkt mit der Computerhardware interagieren müssen, beispielsweise Gerätetreiber, Firmware und das Basic Input/Output System (BIOS). Insbesondere Hochgeschwindigkeitsgeräte, die die Hardwareleistung ausreizen, benötigen oft in Assemblersprache programmierte Treiber und Firmware.

Programmierung von Mikrocontrollern

Assemblersprache wird auch zur Programmierung von Mikrocontrollern genutzt, kleinen, leistungsschwachen Computern, die in einer Vielzahl eingebetteter Systeme (von Spielzeug bis hin zu industriellen Steuerungen) eingesetzt werden. Einige Mikrocontroller haben Speicherkapazitäten von nur einigen hundert Bytes und werden üblicherweise in Assemblersprache programmiert.

Assemblersprache ist das älteste Programmierwerkzeug und hat eine Reihe von Vorteilen, die bei der Programmierung in Maschinencode undenkbar waren. Manchmal wird übrigens die Assemblersprache kurz als Assembler bezeichnet.

Maschinencode und Assemblersprache unterscheiden sich von einem Prozessor zum anderen. Man bezeichnet sie als “Low-Level-Sprachen”, weil sie direkt auf der Hardware arbeiten. Die Konzepte der Berechnung und der Ein-/Ausgabe sind jedoch bei allen Prozessoren gleich. Lassen sich die gemeinsamen Konzepte auf eine Weise ausdrücken, die für den Menschen leichter verständlich ist, erhöht das die Effizienz der Programmierung deutlich — und hier kommen die “High-Level-Sprachen” ins Spiel.

Kompilierte Sprachen

Kompilierte Sprachen sind Programmiersprachen, die entweder in Maschinencode oder in ein Zwischenformat namens *Bytecode* übersetzt werden. Bytecode wird auf dem Zielcomputer von einer *virtuellen Maschine* zur Laufzeit ausgeführt. Die VM übersetzt den Bytecode in den richtigen Maschinencode für jeden Computer. Bytecode ermöglicht es, dass Programme plattformunabhängig sind und auf jedem System mit einer kompatiblen virtuellen Maschine laufen.

Die Übersetzung des in einer höheren Programmiersprache geschriebenen Quellcodes in Maschinencode oder Bytecode erfolgt durch einen *Compiler*. Beispiele für kompilierte Sprachen, die direkt Maschinencode erzeugen, sind C und C++. Sprachen, die Bytecode erzeugen, sind Java und C#.

Die Wahl zwischen Maschinencode und Bytecode hängt von den Anforderungen des Projekts ab, beispielsweise Leistung, Plattformunabhängigkeit und Einfachheit der Entwicklung.

Interpretierte Sprachen

Interpretierte Sprachen sind Programmiersprachen, die von einem *Interpreter* ausgeführt werden, anstatt in Maschinencode kompiliert zu werden. Der Interpreter liest den Quellcode und führt die darin enthaltenen Anweisungen aus. Ein Interpreter ist in der Lage, den Quellcode direkt zu verarbeiten, ohne ihn in ein anderes Dateiformat umzuwandeln. Im Gegensatz zu einem Compiler, der das gesamte Programm in Maschinencode übersetzt, bevor er es ausführt, liest ein Interpreter jede Codezeile und führt sie sofort aus, so dass der Programmierer die Ergebnisse jeder Zeile sehen kann, während sie ausgeführt wird.

Interpretierte Sprachen werden häufig für das *Scripting* verwendet, also für das Schreiben kurzer Programme zur Automatisierung von Aufgaben, für Befehlszeilenschnittstellen und für die Batch- und Job-Steuerung. In interpretierten Sprachen geschriebene Skripte können leicht geändert und ausgeführt werden, ohne dass eine Neukompilierung erforderlich ist, wodurch sie sich gut für Aufgaben eignen, die ein schnelles Prototyping oder eine flexible und schnelle Iteration erfordern. Mit diesen Vorteilen gehen einige potenzielle Nachteile einher, etwa dass ein interpretiertes Programm langsamer läuft als ein entsprechendes kompiliertes Programm.

Beispiele für interpretierte Sprachen sind Python, Ruby und JavaScript. Python wird häufig für wissenschaftliche Berechnungen, Datenanalysen und maschinelles Lernen verwendet, während Ruby bei der Webentwicklung und in Automatisierungsskripten eine große Rolle spielt. JavaScript ist eine clientseitige Skriptsprache, die in Webbrowser eingebettet ist, um dynamische und interaktive Webseiten zu erstellen.

Datenorientierte Sprachen

Datenorientierte Sprachen sind für den effizienten Umgang mit großen Mengen strukturierter oder unstrukturierter Daten konzipiert und bieten eine Reihe von Werkzeugen für die Arbeit mit Datenbanken, Datenstrukturen und Algorithmen für die Datenverarbeitung und -analyse.

Datenorientierte Sprachen kommen insbesondere bei Data Science, Big Data Analytics, maschinellem Lernen und Datenbankprogrammierung zum Einsatz. Sie eignen sich gut für Aufgaben, die die Verarbeitung und Analyse großer Datenmengen umfassen, wie Datenbereinigung und -umwandlung, Datenvisualisierung und statistische Modellierung.

Beispiele für datenorientierte Sprachen sind SQL (*Structured Query Language*), R und MATLAB. SQL ist eine Standardsprache für die Verwaltung relationaler Datenbanken und in Wirtschaft und Industrie weit verbreitet. R ist eine Programmiersprache und -umgebung für statistische Berechnungen und Grafiken und wird häufig in den Bereichen Datenwissenschaft und maschinelles Lernen eingesetzt. MATLAB ist eine numerische Rechenumgebung und Programmiersprache, die in einer Vielzahl von Anwendungen eingesetzt wird, darunter Signalverarbeitung, Bildverarbeitung und Finanzberechnungen.

Programmierparadigmen

Neben den Besonderheiten von Programmiersprachen bestimmen *Programmierparadigmen* den jeweiligen Lösungsansatz. Ein Paradigma kann man sich als grundlegende Strategie vorstellen, mit der wir abhängig von den spezifischen Anforderungen und Bedingungen an eine Aufgabe herangehen.

Ein vergleichbares Beispiel ist der Bau eines Hauses: Ob Maurer die Wände Stein für Stein errichten oder Fertigbetonteile auf der Baustelle zusammengesetzt werden, ist eine grundsätzliche Entscheidung, die von den Anforderungen und Umständen abhängt: Welche Eigenschaften soll das Haus haben? Wo steht es? Ist es mit anderen Häusern verbunden?

In ähnlicher Weise geben Paradigmen die Richtung der Programmierung vor: ob und auf welche Weise beispielsweise ein Softwareprojekt in kleinere, separate Teile zerlegt wird. Jede Programmiersprache eignet sich am besten für ein bestimmtes Paradigma. Daher ist die Wahl des Paradigmas eng mit der Wahl der Programmiersprache verbunden.

Die folgenden Paradigmen sind in der Programmierung üblich:

Objektorientierte Programmierung (OOP)

OOP basiert auf dem Konzept der *Objekte*, die Instanzen von *Klassen* sind, die Daten und Verhalten kapseln. Eine Sprache könnte beispielsweise ein Rechteck als Klasse anbieten, um

dem Programmierer zu helfen, einen Kasten auf dem Bildschirm darzustellen.

OOP konzentriert sich auf die Manipulation von Daten auf Objektebene. OOP macht es einfacher, Code zu schreiben, der wartbar, wiederverwendbar und erweiterbar ist, und wird häufig in Desktop-Software, Videospielen und Webanwendungen verwendet. Beispiele für objektorientierte Programmiersprachen sind Java, C# und Python.

Prozedurale Programmierung

Die prozedurale Programmierung führt Aufgaben durch *Prozeduren* oder Codeblöcke aus, die in einer bestimmten Reihenfolge ausgeführt werden können. Dies macht es einfach, strukturierten Code zu schreiben, der leicht zu erfassen ist, kann aber zu Code führen, der weniger flexibel und schwieriger zu warten ist, wenn Größe und Komplexität des Projekts zunehmen. Beispiele für prozedurale Programmiersprachen sind C, Pascal und Fortran.

Es gibt weitere Ansätze für die Softwareentwicklung, und manche Sprachen sind dafür besser geeignet als andere. Darüber hinaus ermöglichen es Drag-and-Drop-Schnittstellen auch Nicht-Programmierern, Programme zu schreiben, und viele Online-Dienste generieren neuerdings Code durch künstliche Intelligenz, wenn sie Anweisungen in natürlicher Sprache erhalten.

Zusammenfassend lässt sich sagen, dass jedes Programmierparadigma seine Stärken und Schwächen hat und dass die Wahl des Paradigmas von den Anforderungen des Projekts, der Erfahrung und den Vorlieben der Entwickler sowie den Einschränkungen der Plattform und der Entwicklungsumgebung abhängt. Das Verständnis der verschiedenen Paradigmentypen kann helfen, das richtige Paradigma zu wählen sowie besseren und effizienteren Code zu schreiben.

Geführte Übungen

1. Was ist der Zweck von Funktionen?

2. Was ist der Vorteil von Bytecode gegenüber einer Datei mit Maschinencode?

3. Was ist der Vorteil einer Datei mit Maschinencode gegenüber Bytecode?

Offene Übungen

1. Was sind Nachteile der Aufteilung eines Programms in eine große Anzahl von Prozessen oder Tasks?

2. Sie haben mehrere Open-Source-Pakete gefunden, die in verschiedenen Versionen angeboten werden und die Funktionen bieten, die Sie für Ihr Programm benötigen. Was sind einige Kriterien für die Auswahl eines Pakets?

3. Welche anderen Paradigmen der Softwareentwicklung gibt es neben OOP und der prozeduralen Entwicklung und welche Programmiersprache unterstützt den jeweiligen Ansatz am besten?

Zusammenfassung

In dieser Lektion haben Sie gelernt, was Software ist und wie sie mit Hilfe von Programmiersprachen entwickelt wird. Die zahlreichen Programmiersprachen unterscheiden sich nicht nur in ihrer Syntax, sondern zum Beispiel auch in der Verwaltung von Hardwareressourcen oder der Handhabung von Datenstrukturen.

Programmiersprachen unterscheiden sich auch darin, wie der für Menschen lesbare Quellcode von einem Interpreter oder Compiler in den endgültigen Maschinencode umgewandelt wird, den der Computer verarbeitet.

Programmierparadigmen bestimmen die Strategie von Softwareprojekten und damit auch die Wahl geeigneter Programmiersprachen, abhängig von den Anforderungen und der Größe des jeweiligen Projekts.

Antworten zu den geführten Übungen

1. Was ist der Zweck von Funktionen?

   Funktionen kapseln allgemeine Aufgaben, wie etwa die Ausgabe einer Zeichenkette. Mit Hilfe einer Funktion kann Ihr oder ein anderes Programm eine Aufgabe wiederholt ausführen, ohne dass Sie dafür jeweils eigenen Code schreiben müssen.

2. Was ist der Vorteil von Bytecode gegenüber einer Datei mit Maschinencode?

   Die Datei mit Bytecode kann auf vielen verschiedenen Computern ausgeführt werden, wo eine virtuelle Maschine den Code in Maschinencode umwandelt. JavaScript läuft zum Beispiel in vielen Browsern auf vielen verschiedenen Computern.

3. Was ist der Vorteil einer Datei mit Maschinencode gegenüber Bytecode?

   Maschinencode läuft so schnell wie möglich. Bytecode läuft langsamer, weil die virtuelle Maschine ihn in Maschinencode umwandeln muss, während sie den Bytecode ausführt.

Antworten zu den offenen Übungen

1. Was sind Nachteile der Aufteilung eines Programms in eine große Anzahl von Prozessen oder Tasks?

   Wenn ein Programm in Prozesse aufgeteilt ist, müssen diese miteinander kommunizieren. Bei der gemeinsamen Arbeit mit umfangreichen Daten können die Prozesse viel Overhead beim Datenaustausch und beim Schutz der Daten vor gleichzeitigen Änderungen (Race Conditions) erzeugen. Prozesse verursachen auch Overhead, wenn sie gestartet und beendet werden. Je mehr Prozesse es gibt, desto komplexer das Programm und seine Interaktionen, so dass es schwieriger sein kann, Fehler zu finden.

   Das funktionale Paradigma macht es tendenziell einfacher, Programme in viele Prozesse aufzuteilen, da unveränderliche Daten keinen Race Conditions unterliegen.

2. Sie haben mehrere Open-Source-Pakete gefunden, die in verschiedenen Versionen angeboten werden und die Funktionen bieten, die Sie für Ihr Programm benötigen. Was sind einige Kriterien für die Auswahl eines Pakets?

   Überprüfen Sie Fehlerberichte und Sicherheitshinweise für die Pakete, da einige fehlerhaft und sogar unsicher sein können. Manchmal ist die neueste Version nicht die beste, da eine Sicherheitslücke hinzugekommen sein könnte.

   Sehen Sie sich das Forum an, in dem die Entwickler über das Paket diskutieren, um festzustellen, ob es aktiv gewartet wird. Ein wichtiges Kriterium ist vermutlich, dass das Paket langfristig verfügbar und robust ist.

   Testen Sie verschiedene Pakete auf Leistung und Korrektheit.

   Die meisten Pakete sind auf Funktionen in anderen Paketen angewiesen (*Abhängigkeiten*), so dass eine Schwäche in einer der Abhängigkeiten Ihr Programm beeinflussen kann.

3. Welche anderen Paradigmen der Softwareentwicklung gibt es neben OOP und der prozeduralen Entwicklung und welche Programmiersprache unterstützt den jeweiligen Ansatz am besten?

   Neben OOP und prozeduralen Entwicklungsparadigmen gibt es weitere:

   Die funktionale Programmierung legt den Schwerpunkt auf die Verwendung von Funktionen und mathematischen Konzepten wie Lambdas und Closures, um Code zu schreiben, der auf der Auswertung von Ausdrücken und nicht auf der Ausführung von Anweisungen basiert. Funktionale Programmierung behandelt Funktionen vorrangig — so dass sie vom Programm

manipuliert werden können — und betont die *Unveränderlichkeit* oder die Arbeit mit Variablen, die sich nicht mehr ändern können, nachdem sie einmal gesetzt wurden. Das macht es einfacher, den Code zu analysieren und ihn zu testen sowie nebenläufige und parallele Anwendungen zu schreiben. Beispiele für funktionale Programmiersprachen sind Erlang, Haskell, Lisp und Scheme.

Imperative Sprachen konzentrieren sich auf die Anweisungen, die erforderlich sind, um den Fluss der Übergänge des Programms von und zu verschiedenen Zuständen zu steuern.

Deklarative Sprachen beschreiben die zu ergreifenden Maßnahmen und die Logik hinter den Anweisungen. Die Reihenfolge, in der die Anweisungen ausgeführt werden, ist nicht festgelegt. Diese Anweisungen, Funktionsaufrufe und anderen Anweisungen können von Compilern neu geordnet und optimiert werden, sofern die zugrundeliegende Logik erhalten bleibt.

Natürliche Programmierung ist ein Programmierparadigma, das natürliche Sprache oder andere menschenfreundliche Darstellungen verwendet, um das gewünschte Verhalten eines Programms zu beschreiben. Die Idee ist, das Programmieren für Menschen zugänglich zu machen, die keine formale Ausbildung in Informatik haben. Beispiele für natürliche Programmiersprachen sind Scratch und Alice.

051.2 Softwarearchitektur

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 051.2

Gewichtung

2

Hauptwissensgebiete

- Verständnis der Konzepte von Client- und Server-Computing
- Verständnis der Konzepte von Thin und Fat Clients
- Verständnis der Konzepte von Monolithen und Microservices und deren Hauptunterschiede
- Verständnis der Konzepte von Anwendungsprogrammierschnittstellen (APIs)
- Verständnis des Konzepts von Softwarekomponenten und ihrer Integration oder Trennung (Dienste, Module, APIs)

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Clients und Server
- Thin Clients und Fat Clients
- Web-Anwendungen
- Single-Page-Anwendungen
- Monolithische Architekturen
- Microservice-Architekturen
- Anwendungsprogrammierschnittstellen (APIs)
- RESTful APIs

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	051 Grundlagen von Software
Lernziel:	051.2 Softwarearchitektur
Lektion:	1 von 1

Einführung

Das Internet ist allgegenwärtig, und gleiches gilt für mobile und webbasierte Anwendungen. Diese Tools werden von einem großen Teil der Weltbevölkerung genutzt und machen Sofortnachrichten ebenso möglich wie den Online-Kauf von Bergbauausrüstung für große Unternehmen.

Hinter all den scheinbar simplen Schnittstellen und Online-Diensten verbirgt sich eine *Architektur* — eine Struktur interagierender Software, die wir als selbstverständlich betrachten. Man muss ein wenig über diese Architektur wissen, um zu verstehen, wie die Teile des Internets zusammenpassen und wie Software ihren Benutzern all diese Services bietet.

In dieser Lektion sehen wir uns einige Softwarearchitekturen hinter Webanwendungen an, bei denen es sich ja um serverbasierte Software handelt, und wie sie in Systemen eingesetzt werden, die fast jeder kennt.

Server und Clients

Wenn Sie Online-Systeme benutzen, ist es sehr wahrscheinlich, dass Sie irgendwann auf eine Meldung wie die in Beispiel für eine Anzeige, während eine Antwort unterwegs ist gestoßen sind.

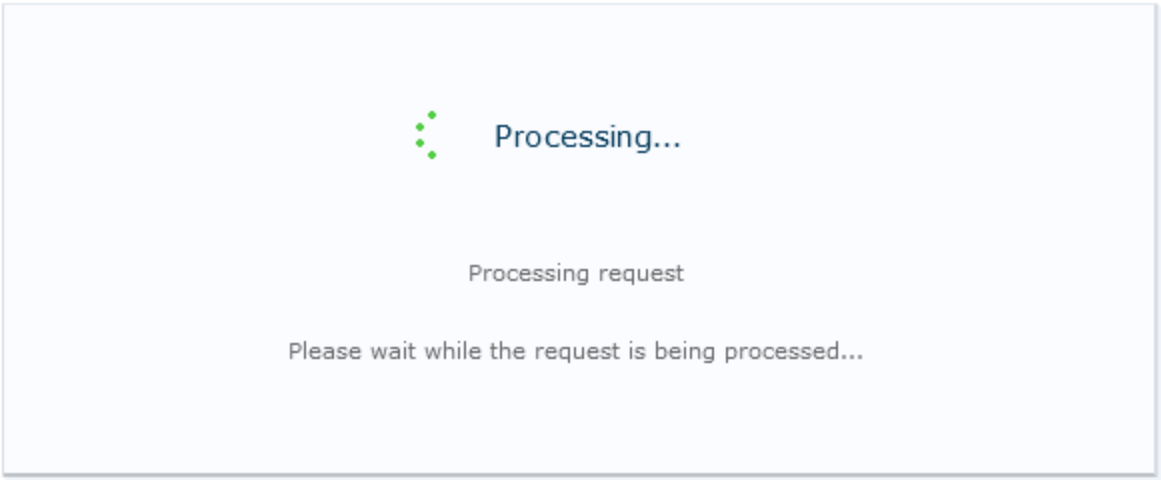

Figure 1. Beispiel für eine Anzeige, während eine Antwort unterwegs ist

Treten wir einen Schritt zurück und betrachten wir den Kontext, in dem diese Meldung erscheint. Nehmen wir an, Sie versuchen, über die Website Ihrer Bank Zugang zu Ihrem Girokonto zu erhalten. Um von Ihrem Laptop auf die Website der Bank zuzugreifen, verwenden Sie eine Software (also eine Anwendung), die als Webbrowser bezeichnet wird, beispielsweise Google Chrome oder Firefox. In diesem Fall sendet der Browser auf Ihrem Computer eine Anfrage an den Rechner, der die Website bereitstellt (hostet). Dieser wird als *Server* bezeichnet. Er ist dafür ausgelegt, rund um die Uhr zu laufen und Anfragen zu bedienen, die aus allen Teilen der Welt eingehen.

Ein Server ist also ein Computer, genau wie der, mit dem Sie arbeiten, Videospiele spielen und Programmieraufgaben erledigen. Es gibt jedoch einen wesentlichen Unterschied: Ein Server wendet normalerweise all seine Ressourcen für die Anwendung auf, die auf ihm läuft — in diesem Beispiel die Webanwendung.

In Beispiel für eine Anzeige, während eine Antwort unterwegs ist hat der Server eine Anfrage von einem Browser erhalten, und die auf dem Server laufende Anwendung verarbeitet die daraus resultierende Operation. Diese Operation könnte eine Datenbankabfrage sein, um die Daten eines Bankkunden der Bank abzurufen, oder die Kommunikation mit einem anderen Server, um einen

Sonderrabatt für den nächsten Kredit des Benutzers zu überprüfen.

In diesem Beispiel bezeichnen wir den Browser, der auf Ihrem Rechner läuft, als *Client-Anwendung* oder einfach *Client*. Der Client interagiert mit dem entfernten Server.

Die Netzwerkkommunikation zwischen Client und Server kann innerhalb eines Unternehmensnetzwerks oder über das weltweite Netzwerk, das wir als Internet bezeichnen, stattfinden. Ein Merkmal der Client-Server-Interaktion ist, dass ein Server mehrere Verbindungen zu mehreren Clients aufbauen kann. Denken Sie an das Beispiel: Die Website einer Bank, die auf einem Server gehostet wird, kann Tausende von Anfragen pro Minute von mehreren Standorten erhalten, wobei jeder Benutzer versucht, auf sein persönliches Bankkonto zuzugreifen.

Nicht alle Szenarien sind so strukturiert, dass ein Browser mit einem Server interagiert, der fast die gesamte Verarbeitung übernimmt. In einigen Fällen ist der Client die primäre Instanz für die Verarbeitung; dieses Konzept wird als *Fat Client* (oder *Thick Client*) bezeichnet, denn der Client speichert und verarbeitet den Großteil der Aufgaben anstatt sich auf die Ressourcen des Servers zu verlassen. In unserem Bankbeispiel ist der Browser jedoch ein *Thin Client*, der sich darauf verlässt, dass der Server Berechnungen ausführt und Informationen über das Netzwerk zurückgibt.

Beispiel für einen Fat Client ist die Desktop-Anwendung eines Videospiels, wo Datenspeicherung und -verarbeitung größtenteils lokal erfolgen, und zwar mit Hilfe von GPU, RAM, CPU und Festplattenspeicher des Computers. Eine solche Anwendung ist nur selten auf einen externen Server angewiesen, insbesondere wenn das Spiel offline gespielt wird.

Beide Ansätze haben Vor- und Nachteile: Bei einem Thick Client ist die Instabilität des Netzwerks weniger problematisch als bei einem Thin Client, der sich auf einen Remote Server verlässt; Software Updates können hingegen aufwendiger sein, und ein Fat Client benötigt mehr Computerressourcen. Bei einem Thin Client können geringere Kosten ein großer Vorteil sein. Bei beiden Client-Typen kann die Bereitstellung persönlicher Daten für die Anwendung eines Drittanbieters ein Problem darstellen.

Webanwendungen

Eine *Webanwendung* ist eine Software, die auf einem Server läuft, Benutzerinteraktionen verarbeitet und von Fat oder Thin Clients über ein Computernetz kontaktiert wird. Nicht alle Websites gelten als Webanwendungen: Einfache statische Webseiten ohne Interaktivität gelten nicht als Webanwendungen, da der Server keine vom Client initiierten Prozesse in einer Anwendung verarbeitet.

Webanwendungen lassen sich grob in zwei Gruppen einteilen: *Single-Page Applications* (SPA) und

Multi-Page Applications (MPA). Eine SPA hat nur eine Webseite, auf der der gesamte Datenaustausch und das Laden von Daten erfolgt, ohne dass der Benutzer auf eine andere Webseite innerhalb der Anwendung umgeleitet werden muss. Eine MPA hat im Gegensatz zur SPA mehrere Webseiten. Eine Datenänderung kann entweder dieselbe Webseite aktualisieren, von der die Aktion ausging, oder den Benutzer auf eine andere Webseite umleiten.

Nehmen wir das vorangegangene Beispiel, in dem ein Benutzer seine letzten Kontobewegungen auf der Internetbanking-Website überprüfen möchte. Stellen Sie sich vor, dass eine Transaktion stattfindet, nachdem die Webseite geladen wurde. Wenn es sich bei der Webanwendung der Bank um eine SPA handelt, wird die neue Transaktion automatisch auf derselben Webseite angezeigt, ohne dass der Benutzer auf eine neue Seite umgeleitet wird. Wenn der Benutzer seine Kredite überprüft, werden die neuen Informationen ebenfalls auf derselben Seite angezeigt, ohne dass der Benutzer auf eine neue Webseite umgeleitet werden muss. Die Änderung der Seite ohne Umleitung des Benutzers sorgt für eine flüssige Navigation.

Bei einer MPA muss der Server, sobald der Benutzer beispielsweise Informationen zu Krediten anfordert, den Benutzer an einen neuen Ort, d.h. eine andere Webseite, weiterleiten.

Ein bekanntes Beispiel für eine SPA ist Google Mail (Gmail), das den Benutzer nicht auf eine neue Seite umleitet, wenn er beispielsweise den Spam-Ordner anklickt; die Anwendung aktualisiert lediglich den spezifischen Teil der Anzeige mit allen Spam-Nachrichten, bleibt aber auf derselben Webseite.

Eine bekannte MPA ist der E-Commerce-Riese Amazon.com, bei dem jeder Artikel auf einer eigenen Webseite zu finden ist.

Ein Vorteil einer MPA gegenüber einer SPA besteht darin, dass Webanalysen viel einfacher zu erfassen und zu messen sind, was für die Optimierung von Internet-Suchergebnissen überaus wichtig ist.

Normalerweise umfasst eine Webanwendung zwei Teile: *Frontend* und *Backend*. Das Frontend ist die Ansichtsschicht, in der Benutzer mit einem Browser interagieren und die Seitenelemente durch Klicken, Auswählen oder Eingaben nutzen. Hier werden die Daten vom Server entgegengenommen, formatiert und dem Benutzer über den Browser angezeigt.

Das Backend ist meist der umfangreichere Teil einer Webanwendung. Es umfasst die Geschäftslogik, die Kommunikationshandler, den Großteil der Datenverarbeitung und die Datenspeicherung. Die Datenspeicherung erfolgt über ein separates Datenbankmanagementsystem, das mit dem Backend verbunden ist.

Frontend und Backend kommunizieren miteinander: Datenanfragen werden vom Frontend an das Backend weitergeleitet, und die vom Backend zurückgegebenen Daten werden vom Frontend

empfangen, formatiert und dem Benutzer angezeigt.

In unserem einfachen Beispiel der Abfrage der letzten Kontotransaktion eines Benutzers wird die Aktion vom Frontend der Anwendung interpretiert, das im Browser auf dem Desktop des Benutzers läuft. Diese Anfrage wird dann über das Internet an das Backend der Anwendung gesendet, das prüft, ob der Benutzer die Aktion ausführen darf, die Daten abruft und die Liste der Transaktionen an das im Browser geladene Frontend zurückschickt. Der Browser formatiert dann die Daten und zeigt sie dem Benutzer an.

Application Programming Interface (API)

Software ist nutzlos, wenn sie nicht mit internen und externen Komponenten kommuniziert. Wie also kommuniziert der Client mit der Webanwendung? Wie sendet das Frontend Daten an das Backend?

Softwareanwendungen kommunizieren miteinander über ein *Application Programming Interface* (API), also eine Programmierschnittstelle, die *Protokolle* der Internetkommunikation nutzt. Protokolle sind Standards und Regeln, die sicherstellen, dass zwei oder mehr Anwendungen Befehle und Daten austauschen.

Der große Vorteil einer API besteht darin, dass sie die verschiedenen Teile einer Anwendung entkoppelt und es ihnen ermöglicht, bei der Verarbeitung von Daten zusammenzuarbeiten. APIs zentralisieren auch den Datenfluss in definierten Kanälen und wirken wie ein Gateway, das sicherstellt, dass alle Komponenten den gleichen Ein- und Ausgang nutzen. APIs sind für die Funktionalität von Webanwendungen von entscheidender Bedeutung, da sie die Interaktion mit dem Benutzer, die Bereitstellung verarbeiteter Informationen, Anfragen zur Datenspeicherung und viele andere Aufgaben ermöglichen. Der Client nutzt eine API beispielsweise, um Aktionen anzufordern, die auf dem Server ausgeführt werden sollen.

Zurück zu unserem Banking-Beispiel: Um sich über eine Webanwendung bei einem Konto anzumelden, gibt der Benutzer in der Regel Daten wie Benutzername und Kennwort in bestimmte Textfelder ein und klickt auf die Schaltfläche “Anmelden”. Der Browser erfasst diese Informationen und ruft eine Backend-API auf. Die Webanwendung, die auf dem entfernten Server läuft, empfängt die Benutzerdaten, validiert den Benutzer, überprüft dessen Zugriffsberechtigung und sendet schließlich eine Antwort an den Browser zurück. Damit der Benutzer mit dem Server kommunizieren kann, müssen sowohl der Client als auch der Server Daten hin- und herschicken — und genau das machen APIs möglich.

Beachten Sie, dass die Webanwendung der Bank keine weiteren sensiblen Informationen preisgibt; sie zeigt dem Benutzer nur die Felder an, die für eine gewünschte Interaktion zulässig und notwendig sind. Der Rest bleibt dem Benutzer verborgen.

Die Kommunikation zwischen APIs kann auf sehr unterschiedlichen Designs und Protokollen basieren. Das *Hypertext Transfer Protocol* (HTTP) ist das bei weitem am häufigsten verwendete Protokoll in Webanwendungen. *Hypertext* ist Text mit Links zu anderen Texten, also das Konzept, das auch den Links in HTML-Webseiten zugrunde liegt. Hyperlinks bilden somit die Grundlage für den Aufbau von Webseiten und deren Darstellung in Browsern.

HTTP wurde für Client-Server-Anwendungen entwickelt, bei denen die Ressourcen von einem Server angefordert und dann über das Netz gemäß einer von HTTP vorgegebenen Struktur an den Client gesendet werden.

Bei einer strukturierten Webanwendung entwerfen die Softwareingenieure die Anwendung in separaten Teilen oder Modulen. Diese Trennung der Zuständigkeiten erlaubt klar definierte Aufgaben und Verantwortlichkeiten, was zur schnellerer Entwicklung und besserer Wartbarkeit führt.

Nehmen wir als Beispiel eine Anwendung mit zwei internen Modulen: eines, das die Geschäftslogik implementiert, und ein anderes, das auf die Integration eines Drittanbieters angewiesen ist. Bei diesem Drittanbieter handelt es sich um ein externes Unternehmen, das seine API für einen bestimmten Zweck zur Verfügung stellt, etwa für die Wettervorhersage. Fällt der Wetterserver aus, ist es unmöglich, Wetterdaten abzurufen; wenn diese Daten für die endgültige Ausgabe aber entscheidend sind, könnte das für den Benutzer vorübergehend ein Problem darstellen, sofern es keine alternative Datenquelle gibt.

Stellen Sie sich nun vor, dass dieser Drittanbieter ersetzt wird und der neue Anbieter dieselbe API auf andere Weise handhabt. Die Trennung der Module bedeutet, dass die Entwickler nur ein Modul aktualisieren müssen. Die Geschäftslogik im anderen Modul muss überhaupt nicht oder nur minimal angepasst werden.

Die Notwendigkeit klarer Prozessstrukturen wirkt sich auch auf das Design der APIs aus, um deren Nutzung zu erleichtern. Das Konzept des *Representational State Transfer* (REST) beschreibt eine Architektur mit einer Reihe von Richtlinien zur Gestaltung und Implementierung des Zugriffs auf die Daten in einer Anwendung.

Es gibt sechs REST-Prinzipien, von denen wir der Einfachheit halber nur drei erläutern, die für diese Lektion am wichtigsten sind.

Client-Server-Entkopplung

Der Client sollte nur den Ressourcen-URI kennen, über den die Kommunikation mit dem Server erfolgt. Dieses Prinzip ermöglicht größere Flexibilität. Wenn beispielsweise die Backend-Seite der Anwendung überlastet ist oder eine größere Änderung in der Architektur einer Backend-Datenbank vorgenommen wird, muss nicht auch das Frontend aktualisiert werden. Es sendet

einfach weiterhin dieselben HTTP-Anfragen an das Backend.

Zustandslosigkeit (Statelessness)

Jede neue Anfrage ist unabhängig von den vorhergehenden. Es ist kein Zufall, dass HTTP für Anwendungen, die den REST-Prinzipien folgen, weit verbreitet ist, da HTTP keine Kenntnis von früheren HTTP-Anfragen hat; für jede neue Anfrage müssen alle notwendigen Informationen gesendet werden, um die Anfrage korrekt zu verarbeiten. Eine Webanwendung, die dieses Prinzip umsetzt, weiß beispielsweise nicht, ob der Client angemeldet (authentifiziert) ist. Daher muss der Client für jede HTTP-Anfrage ein Authentifizierungstoken senden. Der Server kann anhand dieses Tokens überprüfen, ob die Anfrage blockiert oder verarbeitet werden soll.

Einer der Hauptvorteile dieses Prinzips ist die einfachere Skalierung, da der Server Millionen von Anfragen verarbeiten kann, ohne die Benutzerdetails zu überprüfen.

Mehrschichtige Architektur

Die Anwendung besteht aus mehreren Schichten, wobei jede Schicht ihre eigene Logik und ihren eigenen Zweck haben kann, beispielsweise Sicherheit oder Datenerfassung. Der Client weiß möglicherweise nie, wie viele Schichten es gibt oder ob er direkt mit einer bestimmten Schicht innerhalb der Anwendung kommuniziert.

APIs, die den REST-Prinzipien folgen, werden als *RESTful* bezeichnet, und im modernen Web folgen viele Webanwendungen dem REST-Design. Obwohl eine RESTful-API diese Grundsätze nicht mit Hilfe des Protokolls HTTP umsetzen muss, kommt es aufgrund seiner Robustheit, Einfachheit und Allgegenwärtigkeit im World Wide Web fast durchgängig im REST-Modell zum Einsatz.

Architekturtypen

Es gibt Dutzende von Architekturstilen und -standards, die die Strukturen von Webanwendungen organisieren — und wie fast immer in der Computerwelt, gibt es keinen “Gewinner”, sondern eine Reihe von Vor- und Nachteilen für jedes Modell. Ein wichtiges Paradigma ist die so genannte *Microservice-Architektur*, die als Alternative zur älteren *monolithischen Architektur* entstanden ist.

Die Microservice-Architektur ist ein Softwaremodell, das aus mehreren voneinander abhängigen *Services* besteht; diese bilden zusammen die eigentliche Anwendung. Diese Architektur zielt darauf ab, die Codebasis zu dezentralisieren: Mehrere Softwareschichten werden zur besseren Wartung in kleinere Anwendungen aufgeteilt (Microservice-Architektur).

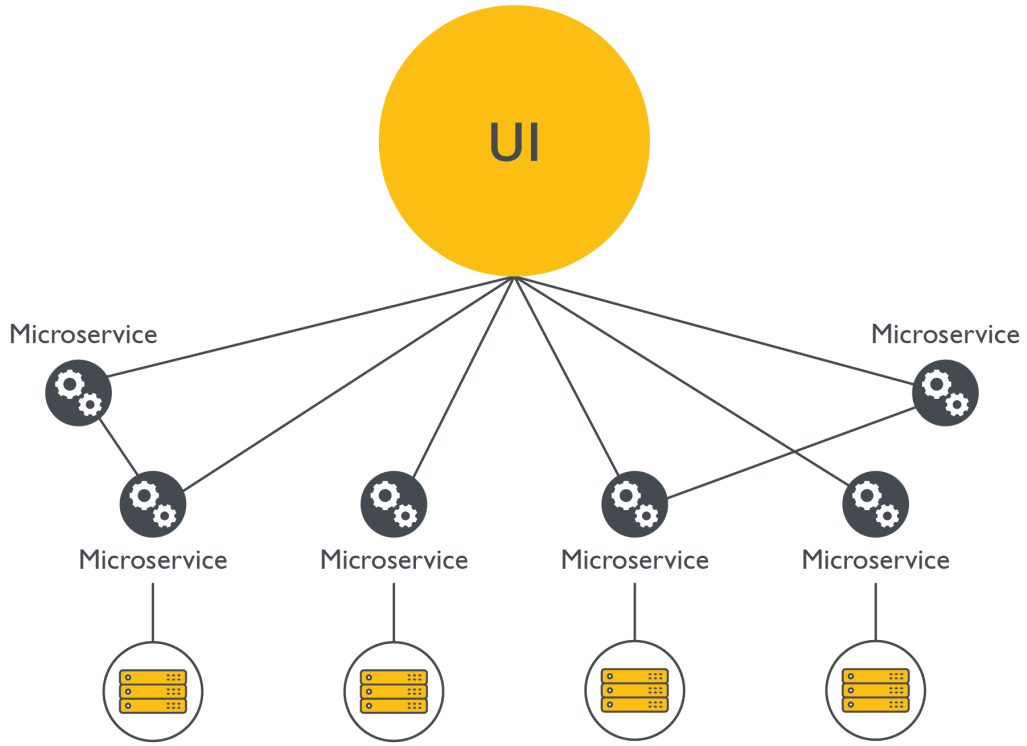

Figure 2. Microservice-Architektur

Im Gegensatz dazu fasst eine monolithische Architektur alle Dienste und Ressourcen der Anwendung in einer einzigen Anwendung zusammen (Monolithische Architektur).

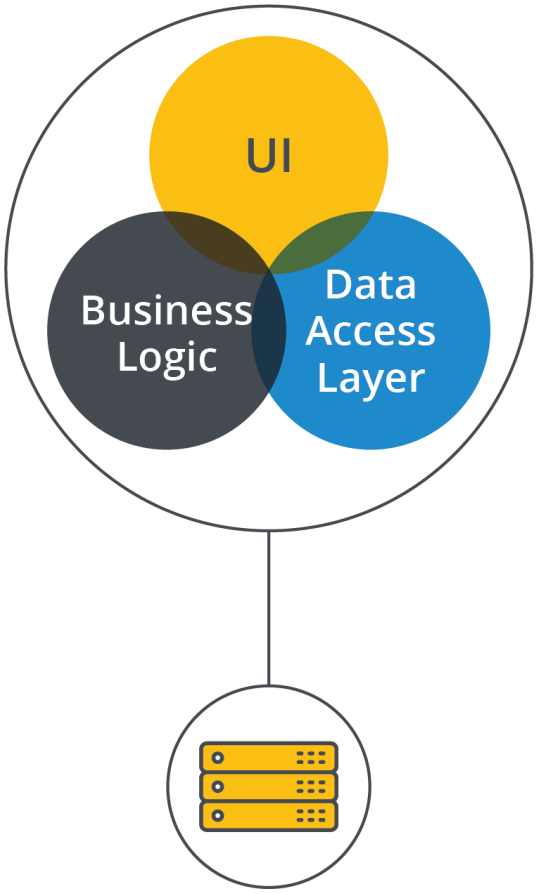

Figure 3. Monolithische Architektur

Die Abbildungen zeigen, dass das Microservice-Modell dezentral angelegt ist und die Anwendung auf mehreren Diensten beruht, von denen jeder seine eigene Datenbank, Codebasis und sogar Serverressourcen hat. Wie der Name schon sagt, sollte jeder Microservice kleiner sein als sein monolithisches Gegenstück, das die Verantwortung für alle Dienste übernimmt.

Die monolithische Anwendung kapselt alle Ressourcen in einer einzigen Einheit. Die gesamte Geschäftslogik, die Daten und die Codebasis sind in einem großen Block zentralisiert, der auch als “Monolith” bezeichnet wird.

Benutzer bemerken kaum, ob eine Webanwendung monolithisch oder als Microservice-Modell angelegt ist; die Wahl sollte transparent sein. Unsere Bankanwendung könnte beispielsweise ein Monolith sein, bei dem sich die gesamte Geschäftslogik in Bezug auf Zahlungen, Transaktionen, Kredite usw. in derselben Codebasis befindet, die auf einem oder mehreren Servern läuft. Folgt die Bankanwendung hingegen Microservice-Paradigma, hat sie wahrscheinlich einen Microservice, der sich mit der Verarbeitung von Zahlungen befasst, und einen weiteren

Microservice, der nur für die Vergabe von Krediten zuständig ist. Der letztgenannte Microservice ruft einen weiteren Microservice auf, um die Wahrscheinlichkeit zu analysieren, mit der der Antragsteller mit der Zahlung in Verzug gerät. Die Anwendung könnte aus Tausenden kleinerer Services bestehen.

Der monolithische Ansatz erfordert höheren Wartungsaufwand, wenn die Anwendung größer wird, insbesondere wenn mehrere Teams in derselben Codebasis programmieren. Angesichts der zentralisierten Softwareressourcen ist es sehr wahrscheinlich, dass ein Team etwas ändert, das den Teil der Anwendung eines anderen Teams beschädigt. Dies kann für größere Teams zu einem echten Problem werden, insbesondere wenn es eine große Anzahl von Teams gibt.

Microservices sind in dieser Hinsicht deutlich flexibler, da jeder Dienst von nur einem Team verwaltet wird. Ein Team kann natürlich auch für mehrere Dienste zuständig sein. Codeänderungen sind leicht durchzuführen und konkurrierende Ressourcen sind kein echtes Problem. Da jeder Dienst mit anderen verbunden ist, kann sich jeder Fehler negativ auf die gesamte Anwendung auswirken. Da außerdem mehrere Datenbankinstanzen, Server und externe APIs miteinander kommunizieren, ist die Ausfallsicherheit der gesamten Anwendung nur so gut wie ihr schwächster Microservice.

Ein Vorteil des monolithischen Ansatzes ist die zentrale Datenquelle, die die Vermeidung von Datenduplizierung erleichtert. Der Ansatz reduziert auch den Verbrauch von Cloud-Ressourcen, da ein größerer Server weniger Ressourcen benötigt als mehrere dezentrale Server. Eine Microservice-Anwendung von etwa gleicher Größe belastet die Cloud stärker.

Geführte Übungen

1. Welche sind die Hauptunterschiede zwischen einem Fat und einem Thin Client?

2. Ist die Annahme richtig, dass jede Website eine Webanwendung ist?

3. Was ist das REST-Modell?

4. Was ist das bevorzugte Modell für die Entwicklung großer, moderner Webanwendungen mit mehreren Entwicklungsteams? Warum?

5. Welches ist das am häufigsten verwendete Protokoll für den Datenaustausch zwischen Webanwendungen?

6. Nennen Sie zwei Nachteile von Multi-Page Anwendungen gegenüber Single-Page Anwendungen.

7. Beschreiben Sie einen Vorteil eines monolithischen Systems gegenüber einem Microservice-System und einen Vorteil, den das Microservice-System gegenüber einem monolithischen System hat.

Offene Übungen

1. Im Jahr 2021 landet der NASA Perseverance Rover auf dem Mars und soll unter anderem herausfinden, ob es jemals Leben auf dem Mars gab. Obwohl der Rover hier auf der Erde aus der Ferne gesteuert werden könnte, kann er sich in den meisten Situationen auch selbst steuern. Warum ist es eine gute Idee, einen solchen Rover als Fat Client zu entwerfen?

2. Stellen Sie sich ein modernes, selbstfahrendes Auto vor, das eine Verbindung zu einem externen Server herstellt, um Daten auszutauschen. Sollte es ein Fat oder ein Thin Client sein?

Zusammenfassung

In dieser Lektion wurden die Kernkonzepte der Softwarearchitektur für Webanwendungen erläutert. Es wurde erklärt, wie diese üblicherweise strukturiert und organisiert sind und worin die Hauptunterschiede zwischen monolithischen und Microservice-Modellen bestehen. Wir haben das Client-Server-Konzept sowie die Grundlagen der Kommunikation von Webanwendungen zwischen Clients und anderen Softwareprogrammen behandelt.

Antworten zu den geführten Übungen

1. Welche sind die Hauptunterschiede zwischen einem Fat und einem Thin Client?

   Ein Fat Client benötigt keine ständige Verbindung zu einem entfernten Server, der wichtige Informationen an den laufenden Client zurückgibt. Der Thin Client ist in hohem Maße auf die von einer externen Quelle verarbeiteten Informationen angewiesen. Ein weiterer Unterschied besteht darin, dass ein Fat Client für den Großteil der Datenverarbeitung verantwortlich ist und daher mehr Rechenressourcen benötigt als sein Thin Pendant.

2. Ist die Annahme richtig, dass jede Website eine Webanwendung ist?

   Es gibt Websites, die keine Softwareanwendungen sind. Eine Webanwendung interagiert mit dem Benutzer, der Daten eingeben und Webfunktionen in Echtzeit nutzen kann. Einfache Websites, wie etwa Werbung für eine Veranstaltung, die wie ein Webbanner funktioniert, sind keine Webanwendungen. Diese nicht interaktiven Websites sind einfacher zu pflegen und erfordern nur geringe Rechenressourcen für das Hosting und die Bereitstellung der Webseiten. Eine Webanwendung erfordert viel mehr Rechenressourcen, robustere Server und Funktionen für den Umgang mit Benutzern, wie beispielsweise Zugangsbeschränkungen und permanente Datenspeicherung.

3. Was ist das REST-Modell?

   Das REST-Modell ist ein Softwarearchitekturmodell, das Anwendungen einen Entwicklungsleitfaden für bessere Benutzerfreundlichkeit, Klarheit und Wartungsfreundlichkeit bietet. Eines der in den REST-Leitlinien dargelegten Prinzipien ist die *Schichtenarchitektur*, die in erster Linie dem Zusammenhalt und der Verringerung von Abhängigkeiten der verschiedenen internen Komponenten der APIs dient.

4. Was ist das bevorzugte Modell für die Entwicklung großer, moderner Webanwendungen mit mehreren Entwicklungsteams? Warum?

   Das Microservice-Softwaremodell bietet einen flexiblen Rahmen, in dem Teams an derselben Softwareanwendung zusammenarbeiten, was die Gleichzeitigkeit für zwei oder mehr Teams, die eine große Webanwendung pflegen, erleichtert. Da das Framework dezentralisiert ist, kann jedes Team einen bestimmten Geschäftsbereich aktualisieren, ohne andere Komponenten aktualisieren zu müssen.

5. Welches ist das am häufigsten verwendete Protokoll für den Datenaustausch zwischen Webanwendungen?

HTTP ist das am häufigsten verwendete Protokoll für den Austausch von Daten und Befehlen zwischen Servern und Clients.

6. Nennen Sie zwei Nachteile von Multi-Page Anwendungen gegenüber Single-Page Anwendungen.

   Bei einer Multi-Page Anwendung werden alle Elemente der Webseite neu geladen, sobald der Benutzer bestimmte Aktionen auslöst, anstatt nur die geänderten Elemente zu aktualisieren. Darunter leidet die Performance. Ein weiterer Nachteil einer MPA ist die trägere Benutzerinteraktivität, da jedes Laden einer Seite die Benutzerfreundlichkeit mindert. Im Gegensatz dazu können visuelle Übergänge bei einer SPA glatter sein.

7. Beschreiben Sie einen Vorteil eines monolithischen Systems gegenüber einem Microservice-System und einen Vorteil, den das Microservice-System gegenüber einem monolithischen System hat.

   Ein monolithisches System kann die Datenverwaltung erleichtern, da sich die Daten in einer großen Datenbank befinden, anstatt in mehreren Datenbanken verstreut zu sein. Eine Microservice-Anwendung hingegen kann die Codeentwicklung und -wartung verbessern: mehrere Teams können an verschiedenen Geschäftslogiken arbeiten, ohne den Fortschritt anderer Teams zu blockieren.

Antworten zu den offenen Übungen

1. Im Jahr 2021 landet der NASA Perseverance Rover auf dem Mars und soll unter anderem herausfinden, ob es jemals Leben auf dem Mars gab. Obwohl der Rover hier auf der Erde aus der Ferne gesteuert werden könnte, kann er sich in den meisten Situationen auch selbst steuern. Warum ist es eine gute Idee, einen solchen Rover als Fat Client zu entwerfen?

   Die Zeit, die ein Kommunikationssignal braucht, um von der Erde gesendet und auf dem Mars empfangen zu werden, kann je nach Position der beiden Planeten variieren, aber bis zu zwanzig Minuten betragen. Daher ist es unmöglich, einen entfernten Rover in Bewegung zu kontrollieren, insbesondere in unerwarteten Situationen. Im Idealfall sollte sich der Rover in den meisten Situationen selbst steuern. Dies wird durch Training mit künstlicher Intelligenz (KI) erreicht (maschinelles Lernen), so dass der Rover unabhängiger von manuellen Befehlen wird. Um weniger auf Signale aus der Ferne angewiesen zu sein, ist der Rover so konzipiert, dass er über eigene Ressourcen verfügt und den Großteil der Rechenprozesse lokal ausführt, was der Definition eines Fat Client entspricht.

2. Stellen Sie sich ein modernes, selbstfahrendes Auto vor, das eine Verbindung zu einem externen Server herstellt, um Daten auszutauschen. Sollte es ein Fat oder ein Thin Client sein?

   Ein autonomes Fahrzeug könnte die aufwendige Datenverarbeitung an einen externen und zuverlässigen Server delegieren, aber dieser wäre anfällig für Offline-Perioden, wenn kritische Datenverarbeitung erforderlich ist. Daher ist es zwingend erforderlich, dass das autonome Fahrzeug den Großteil der Aufgaben verarbeitet — und das erfordert, dass es ein Fat Client mit mehrfacher Redundanz ist.

051.3 On-Premises und Cloud Computing

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 051.3

Gewichtung

1

Hauptwissensgebiete

- Verständnis der Konzepte von On-Premises und Cloud-Computing
- Verständnis der gängigen Cloud-Betriebsmodelle
- Verständnis der gängigen Arten von Cloud-Diensten
- Verständnis der wichtigsten Vorteile und Risiken von Cloud Computing und On-Premises IT-Infrastruktur

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Cloud Computing
- On-Premises IT-Infrastruktur
- Rechenzentrum
- Öffentliche, private und hybride Clouds
- Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS)
- Kostenmodelle
- Sicherheit
- Dateneigentum
- Verfügbarkeit von Diensten

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	051 Grundlagen von Software
Lernziel:	051.3 On-Premises und Cloud Computing
Lektion:	1 von 1

Einführung

Die Cloud ist heute in aller Munde, und Unternehmen prüfen die Dienste verschiedener Anbieter — darunter auch großer Unternehmen wie Amazon und Microsoft — um zu sehen, was die Cloud bietet.

Aber die Idee des Cloud Computing entstand erst Ende des ersten 2000er-Jahrzehnts, und der Begriff ist so vage, dass manche, darunter die Free Software Foundation, von der Verwendung des Begriffs abraten.

Cloud Computing fasst jedoch einige nützliche Konzepte zusammen und ist ein wichtiger Trend im modernen Computing. Diese Lektion befasst sich damit, was die Cloud ist und wie sie sowohl technisch als auch wirtschaftlich funktioniert. Wir schauen uns Vorteile und Risiken der Cloud an werden dabei erkennen, warum dieses Thema mit Open Source verbunden ist.

On-Premises und Cloud Computing

Wenn Sie in einem Unternehmen arbeiten oder studieren, das mehr als nur ein paar Computersysteme hat, verfügt es mit ziemlicher Sicherheit über ein *Rechenzentrum*: einen

separaten Raum, in dem die Server untergebracht sind, in der Regel gesichert und klimatisiert. Ein Rechenzentrum "vor Ort" (engl. *on-premises* oder *on-premise*) ist einfach eines, das ein Unternehmen für die eigene Nutzung unterhält.

Es gibt mehrere Alternativen zum Betrieb eines Rechenzentrums vor Ort. Jahrzehntelang, vor dem Trend, den wir als Cloud Computing bezeichnen, richteten Unternehmen Rechenzentren ein, in denen Computer im Auftrag von Kunden gehostet wurden. Man konnte also beispielsweise fünf Server von einem Unternehmen lizenzieren, dem Unternehmen verschiedene Spezifikationen für CPU, Arbeitsspeicher und Speicherplatz nennen und dann Software auf diese Server hochladen. Dieses Geschäftsmodell wird als *Remote Hosting* bezeichnet.

Remote Hosting bietet viele Vorteile: Ein Unternehmen lagert das administrative Fachwissen für den Kauf und die Einrichtung von Servern an den Dienstleister für Remote Hosting aus, der als Großkunde die Hardware zudem günstiger einkauft. Mit anderen Worten: Man meidet die Verantwortung für eine IT-Infrastruktur vor Ort. Das Unternehmen für Remote Hosting gewährleistet die physische Sicherheit und nimmt den Kunden auch diese Sorge. Und schließlich ist auch die Einrichtung eines neuen Servers bei einem Unternehmen für Remote Hosting viel schneller als Kauf, Versand und Einrichtung des Servers vor Ort.

Ein Unternehmen kann auch ein Rechenzentrum vor Ort betreiben und gleichzeitig weitere Server in der Remote-Umgebung lizenzieren, um nach Ausfällen die Infrastruktur wieder herzustellen oder zusätzliche Rechenleistung in Zeiten hoher Auslastung bereitzustellen.

Wichtig ist, dass Remote Computing ein schnelles und zuverlässiges Netzwerk voraussetzt. Wir werden dieses Thema zusammen mit anderen Vorteilen und Schwächen in einem anderen Abschnitt behandeln.

Alle Vorteile des Remote Hosting gelten auch für das Cloud Computing. Der technische Unterschied zum Cloud Computing besteht darin, dass für ein Unternehmen kein eigener physischer Rechner zur Verfügung steht, sondern dass der Cloud-Anbieter auf jedem physischen System über eine zusätzliche Softwareschicht mehrere Systeme für mehrere Kunden betreibt, sogenannte *virtuelle Maschinen*.

Ein Cloud-Dienst betreibt also ein Rechenzentrum wie jedes andere Unternehmen, bedient aber andere Unternehmen statt nur sich selbst. Das Rechenzentrum umfasst Tausende physische Computer. Auf jedem physischen Computer läuft ein Betriebssystem (in der Regel *Hypervisor* genannt), das mehrere virtuelle Maschinen unterstützt. Jede virtuelle Maschine kann schnell erzeugt und gelöscht werden. Jede virtuelle Maschine unterstützt ein Betriebssystem, das von einem Client (Cloud Computing) ausgeführt wird.

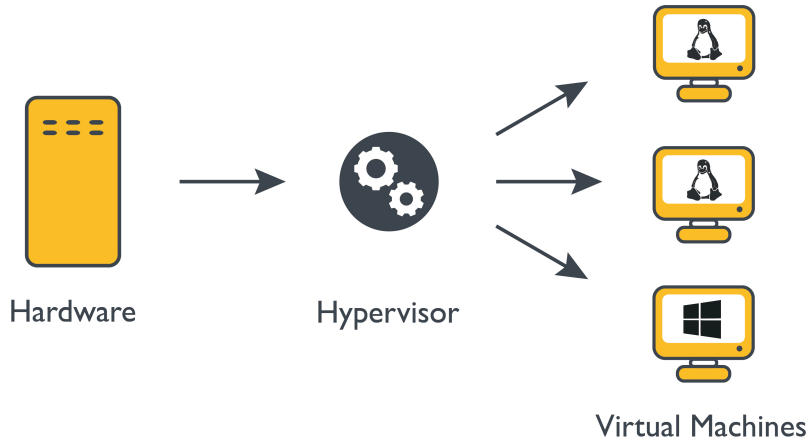

Figure 4. Cloud Computing

Wo kommen nun freie und Open Source Software ins Spiel? Wenn ein Unternehmen eine große Anzahl von Computern betreibt und Betriebssysteme im laufenden Betrieb bereitstellt, ist es wichtig, sich nicht mit Lizenzen zu verzetteln. Es gibt zwar Lizenzmodelle für proprietäre Betriebssysteme in der Cloud, aber sie sind komplizierter als der Betrieb einer virtuellen Maschine und eines Open-Source-Betriebssystems. Open Source ist in der Regel auch kostenlos.

Die Anfänge des Cloud Computing werden in der Regel mit der Einführung der Amazon Web Services (AWS) durch Amazon.com im Jahr 2006 in Verbindung gebracht. Inzwischen gibt es Dutzende Cloud-Unternehmen, darunter Angebote von so großen Anbietern wie Microsoft, Google, Alibaba und IBM. AWS ist nach wie vor das größte Angebot. Die Anbieter konkurrieren bei der Entwicklung neuer Funktionen und Dienste, da sie alle stark auf Kosten und Zuverlässigkeit setzen.

Die Vorteile des Cloud Computing bauen auf denen des Remote Computing auf. Die Kosten sind niedriger, weil ein physisches System viele Server für viele Kunden betreiben kann und ständig ausgelastet ist. Ein Kunde, der schnell mehr Rechenleistung für eine Nutzungsspitze benötigt, kann in Sekundenschnelle neue Systeme erzeugen, und die Systeme lassen sich automatisch über eine Programmierschnittstelle (API) verwalten. Auch hier kommen wir noch auf Vorteile und Risiken zu sprechen.

Modelle für den Cloud-Betrieb

Bevor wir auf die Betriebsmodelle eingehen, sei erwähnt, dass viele Unternehmen Cloud-Modelle eingeführt haben, die die Art und Weise, wie sie programmieren und Dienste anbieten, völlig

verändern. Anstatt eine Anwendung ein- oder zweimal im Jahr zu aktualisieren, sind nun schnelle Aktualisierungen möglich, weil die Cloud es erlaubt, virtuelle Maschinen herunterzufahren und neue Maschinen fast zeitgleich mit der neuen Version der Anwendung zu starten. Ein Unternehmen kann auch schnell nach oben oder unten skalieren, weshalb Anwendungen gerne in viele modulare Teile (*Microservices*) aufgeteilt werden.

In diesem Abschnitt konzentrieren wir uns jedoch auf gängige Cloud-Modelle.

Das Kostenmodell für die Cloud unterscheidet sich stark von den Kosten für ein Rechenzentrum vor Ort. On-Premises müssen zunächst Server gekauft werden, außerdem fallen laufende Kosten für Strom, Klimatisierung und Verwaltung an. Diese Kosten entfallen, wenn man Systeme von einem Cloud-Anbieter lizenziert. Stattdessen werden genutzte Kapazitäten berechnet. Cloud-Anbieter unterteilen die Computernutzung in Zeiträume und berechnen diese. Außerdem berechnen sie die Datenmenge, die auf den Systemen gespeichert werden.

Bisher haben wir über Anbieter gesprochen, die Kunden Rechenleistung anbieten, die sogenannte *Public Cloud*. Es gibt aber auch die *Private Cloud*: So betreiben einige große Unternehmen ihre eigenen Rechenzentren vor Ort wie eine Cloud. Sie stellen ihre Dienste nur ihren eigenen Abteilungen oder Unterabteilungen zur Verfügung, behandeln aber jede dieser Abteilungen wie einen Kunden eines Cloud-Anbieters. Das Rechenzentrum verfolgt, wie viel Rechenzeit, Daten usw. von jeder Abteilung genutzt werden, und stellt diese Nutzung in Rechnung.

Viele Unternehmen nutzen auch mehrere Cloud-Dienste, beispielsweise um sich vor Ausfällen eines Anbieters zu schützen, Daten in einer bestimmten geografischen Region zu speichern oder spezielle Funktionen eines bestimmten Anbieters zu nutzen. Außerdem ist es üblich, sowohl ein Rechenzentrum vor Ort als auch Server in der Cloud zu unterhalten, was als *Hybrid Computing* bezeichnet wird.

Ein Kunde, der sich für einen Cloud-Service anmeldet, kann wählen, in welchen geografischen Regionen die Dienste laufen. Amazon bietet beispielsweise derzeit Regionen im Westen und Osten der USA, mehrere Regionen in Europa und weitere Regionen in allen Teilen der Welt an. Normalerweise würden Sie die Region wählen, die Ihnen am nächsten liegt. Aber viele Unternehmen möchten in mehreren Regionen arbeiten, weil sie international tätig sind. Zudem müssen Unternehmen manchmal Daten an einem bestimmten Ort aufbewahren, um etwa konform zur europäischen Datenschutzverordnung (GDPR) oder dem chinesischen Gesetz zum Schutz persönlicher Daten (PIPL) zu sein.

Jede Region ist in der Regel in *Zonen* oder *Verfügbarkeitszonen* unterteilt. Der Betrieb in mehreren Zonen wird empfohlen, falls ein Schaden den Ausfall einer Zone verursacht.

Obwohl die Cloud für die gemeinsame Nutzung physischer Computer durch mehrere Kunden bekannt ist, können einige Anbieter auch einzelne Rechner Kunden exklusiv bereitstellen, die dies

aus Gründen der Sicherheit benötigen. Da keine andere Organisation den Computer nutzt, ist es für den Kunden möglicherweise sicherer, wenn er sensible Dienste ausführt und Daten in die Cloud hochlädt. Diese Option macht das Cloud Computing dem ursprünglichen Remote Hosting ähnlich.

Cloud Services

Cloud Computing sieht auf verschiedenen Ebenen sehr unterschiedlich aus und richtet sich an verschiedene Benutzergruppen.

Infrastructure as a Service (IaaS) ist die Kategorie, mit der sich Systemadministratoren in der Regel befassen. IaaS stellt lediglich Hardware und Software zur Verfügung, die virtuelle Maschinen unterstützen. Es obliegt den Systemadministratoren des Kunden, ein Betriebssystem und die gewünschten Anwendungen auf die virtuelle Maschine zu laden. Der Systemadministrator handhabt fast alles auf dieselbe Weise wie in einem Rechenzentrum vor Ort.

Platform as a Service (PaaS) ist ein neuerer Ansatz, den vor allem Programmierer nutzen. Hier muss sich der Programmierer nicht um das Betriebssystem kümmern und auch nicht die Bibliotheken laden, die das Programm verwendet. All das stellt der Cloud-Anbieter bereit. Der Programmierer lädt lediglich Funktionen hoch, die auf der Plattform ausgeführt werden. Ein verwandtes Konzept ist das *serverlose* Computing.

Software as a Service (SaaS) ist eine Anwendung, die auf einem Cloud-System läuft. Wenn Sie sich bei einer Social Media Website anmelden, einen Artikel in einem Online Shop bestellen, eine Webseite aufrufen, um Ihre Arbeitszeit in ein Job-Tracking-System einzutragen, oder ein Formular auf der Seite einer Behörde ausfüllen, nutzen Sie SaaS. Der Großteil der Anwendung läuft auf dem entfernten System, und der einzige Teil der Anwendung, der auf Ihrem Computer läuft, ist die von Ihrem Browser angezeigte Webseite.

Database as a Service (DaaS) ist oft eine Ergänzung zu den zuvor genannten Services. Ein Datendienst, S3 von Amazon, war eigentlich das erste Cloud-Angebot. DaaS kann einfach eine Instanz eines beliebten Datenbankservers wie MySQL, Oracle oder MongoDB sein, die in der Cloud läuft. Große Cloud-Anbieter bieten auch eigene Datenbanken an, die nur in ihren Cloud-Angeboten laufen. In jedem Fall lesen und schreiben Kunden die Datenbank, als würden sie lokal auf ihren Systemen laufen.

Es gibt weitere Varianten dieser Services, die von einigen Unternehmen angeboten werden, wie etwa Security as a Service.

Vorteile und Risiken von Cloud Computing und On-Premises IT-Infrastruktur

Bevor wir uns genauer mit Cloud Computing befassen, eine Analogie: Der Betrieb eines Rechenzentrums vor Ort ist wie der Kauf eines Hauses. Wird der Keller überschwemmt oder fällt der Heizkessel aus, müssen Sie jemanden finden, der den Schaden behebt. Im Gegensatz dazu entsprechen Remote Hosting und Cloud Computing dem Mieten einer Wohnung: Der Vermieter ist dafür verantwortlich, dass der Heizkessel funktioniert. Außerdem können Sie in der Cloud schnell Datenlasten hinzufügen und entfernen, so wie Sie die Wohnung schneller wechseln können als das Haus, das Sie besitzen.

In einer Wohnung stellt der Vermieter vielleicht sogar Geräte und Möbel zur Verfügung. In unserer Analogie entspricht dies den zahlreichen Diensten der Cloud-Anbieter, wie Datenbanken und Analysen.

Schauen wir uns Vorteile und Risiken der Cloud an, die statt oder ergänzend zu einem eigenen Rechenzentrum genutzt wird.

Flexibilität ist wahrscheinlich der ausschlaggebende Grund für einen Wechsel in die Cloud. Als Einzelhändler, der in der Vorweihnachtszeit mehr Server betreiben muss, oder Steuerberater, der den größten Teil seines Geschäfts in der Steuersaison abwickelt, werden Sie die Cloud nutzen wollen, um jederzeit neue Server in Betrieb nehmen und deren virtuelle Maschinen später löschen zu können.

Die *Kosten* können in der Cloud aus mehreren Gründen niedriger sein. Sie teilen sich einen physischen Server mit vielen anderen Anwendungen, so dass die Rechner effizienter genutzt werden. Da Cloud-Anbieter groß sind, nutzen sie Skalierungseffekte bei Einkauf, Verwaltung, Kühlung und anderen Infrastrukturanforderungen. Schließlich sind die Kunden von vielen administrativen Aufgaben befreit — obwohl die Systemadministration keineswegs obsolet wird. Kunden benötigen immer noch Systemadministratoren, um Software (in der Cloud als *Instanzen* bezeichnet) zu erstellen und hochzuladen, Benutzer zu autorisieren und andere Aufgaben im Zusammenhang mit dem Geschäftsbetrieb zu erledigen. Systemadministratoren müssen die API des Anbieters und die Regeln für die Nutzung des Dienstes kennenlernen, ein Schulungsaufwand, der unbedingt zu berücksichtigen ist.

Darüber hinaus ist im Blick zu behalten, wie intensiv Sie die Cloud nutzen. Es kann schwierig sein, die verbrauchte Rechenleistung zu überblicken, wenn man Server schnell hochfahren kann, vor allem wenn Sie die Skalierung automatisieren. Am Ende eines Abrechnungszeitraums könnte Sie eine unangenehm hohe Rechnung überraschen.

Ist die Cloud CO_2-effizienter als der Betrieb eigener Rechner? Untersuchungen haben ergeben,

dass Cloud-Anbieter ihre Systeme deutlich effizienter betreiben können. Aber die Kommunikation mit diesen Systemen geschieht über ein Netzwerk, was viel Strom für die Versorgung aller Netzwerkgeräte erfordert. Leider vergrößert die Cloud also unseren CO_2-Fußabdruck.

Die *Verfügbarkeit* von Diensten ist in der Cloud oft besser. Das eigene Rechenzentrum vor Ort ist anfällig für eine Reihe von Problemen, von Naturkatastrophen bis hin zu internen Saboteuren. Aber auch Rechenzentren in der Cloud fallen aus. Daher sollten Sie die verschiedenen Verfügbarkeitszonen nutzen und das Risiko streuen. Es gibt Tools, mit denen Sie Dienste von einer ausgefallenen Zone in eine funktionierende wechseln.

Wenn Sie Dienste eines Cloud-Anbieters nutzen, beispielsweise eine Datenbank in der Cloud, sind Sie anfällig für Fehler in diesem Dienst; aber natürlich können Sie auch von Fehlern der Software betroffen sein, die Sie auf Ihr System laden.

Ein größeres Risiko bei der Nutzung externer Dienste ist der sogenannte Lock-in. In der Regel gibt es automatische Konvertierungstools, mit denen Sie Ihre Daten aus dem System des Anbieters in ein neues System übertragen können, aber das Tool erledigt diese Aufgabe möglicherweise nicht vollständig.

Die *Sicherheit* kann in der Cloud höher sein, weil die Mitarbeiter des Anbieters wahrscheinlich fachkundiger sind als Ihr eigenes Personal. Andererseits sind Cloud-Anbieter groß und bekannt — und damit beliebte Angriffsziele. Außerdem stellt das Hinzufügen einer zusätzlichen Software — des Hypervisors, der die virtuellen Maschinen steuert — eine zusätzliche potenzielle Gefahr dar, denn Schwachstellen werden auch in Hypervisoren gefunden.

Obwohl der Kunde rechtlich gesehen Eigentümer seiner Daten bleibt, macht die Speicherung der Daten in der Cloud sie theoretisch angreifbarer. Normalerweise verschlüsselt der Kunde die Daten, um sie im Falle eines Einbruchs zu schützen. Datenschutzbestimmungen, wie die bereits erwähnte GDPR, verlangen, dass die Daten in einem Rechenzentrum in einer Region gespeichert werden, die als sicher gilt.

Die meisten Sicherheitsangriffe beginnen auf Benutzerebene, beispielsweise durch das Versenden von E-Mails mit Malware an einen ahnungslosen Mitarbeiter, und dabei spielt es keine Rolle, ob Sie Ihr System vor Ort oder in der Cloud betreiben. Ein böswilliger Eindringling, der das Konto eines Mitarbeiters übernimmt, wird jedoch nicht viel weiter kommen, es sei denn, er kann Schwachstellen in Ihren Servern ausnutzen; auch hier ist es nicht eindeutig, ob die Nutzung der Cloud einen großen Unterschied macht, da die meisten Schwachstellen in der Software und nicht im Cloud-Dienst zu finden sind.

Schließlich sollten Sie auch Ihre Kosten für Bandbreite und Netzwerk bedenken. Kunden und Mitarbeiter kommunizieren mit Servern, die Hunderte von Kilometern entfernt sein können. Ist die Netzwerkverbindung unzuverlässig oder langsam, wird die Leistung von Cloud-Servern

schlechter sein als die eines Rechenzentrums vor Ort. Heute verbindet sich jeder mit Remote-Mitarbeitern, SaaS-Diensten und anderen Systemen, die geografisch weit entfernt sind — die Performance des Netzwerks wirkt sich darum auf fast alle Arbeitsschritte aus, sei dies in der Cloud oder nicht.

Geführte Übungen

1. Warum wird ein physischer Rechner in einem Cloud Center effizienter genutzt als ein Rechner in einem Rechenzentrum vor Ort?

2. Was ist eine hybride Cloud?

3. Welche Art von Cloud Computing erfordert am häufigsten das Eingreifen eines Systemadministrators auf Kundenseite?

4. Wie sollten Sie Ihren Dienst vor Ausfällen schützen, wenn Sie einen Cloud-Anbieter nutzen?

Offene Übungen

1. Vergleichen Sie die verschiedenen Kosten, die entstehen, wenn Sie Ihre Server in einer Cloud betreiben, mit den Kosten für den Betrieb vor Ort.

2. Sie sind vom Nahen Osten aus tätig, haben aber viele Kunden in Europa und im Fernen Osten. Beschreiben Sie, wo Sie Ihre Dienste im Rahmen eines Cloud-Angebots platzieren würden.

Zusammenfassung

In dieser Lektion haben Sie erfahren, wie Cloud Computing funktioniert und welche Vorteile die Nutzung der Cloud gegenüber dem Betrieb von Systemen in Ihrem eigenen Rechenzentrum hat. Sie haben verschiedene Geschäfts- und Kostenmodelle kennengelernt, einschließlich der Unterschiede zwischen öffentlichen, privaten und hybriden Clouds. Sie haben auch die verschiedenen Haupttypen von Cloud-Angeboten kennengelernt und wissen, wofür sie jeweils eingesetzt werden.

Antworten zu den geführten Übungen

1. Warum wird ein physischer Rechner in einem Cloud Center effizienter genutzt als ein Rechner in einem Rechenzentrum vor Ort?

   In der Cloud kann jeder Rechner mehrere Instanzen von Betriebssystemen ausführen und sogar von verschiedenen Kunden hochgeladene Instanzen ausführen. Daher wird der Computer intensiver genutzt.

2. Was ist eine hybride Cloud?

   Eine hybride Cloud nutzt sowohl Rechenzentren bei einem Cloud-Anbieter als auch ein oder mehrere Rechenzentren vor Ort.

3. Welche Art von Cloud Computing erfordert am häufigsten das Eingreifen eines Systemadministrators auf Kundenseite?

   Bei Infrastructure as a Service (IaaS) übernimmt der Kunde die Systemverwaltung wie das Erstellen und Hochladen von Instanzen des Betriebssystems oder von Anwendungen.

4. Wie sollten Sie Ihren Dienst vor Ausfällen schützen, wenn Sie einen Cloud-Anbieter nutzen?

   Wählen Sie mehrere Zonen in jeder Region, in der Sie Ihren Dienst betreiben, denn es ist sehr unwahrscheinlich, dass mehrere Zonen gleichzeitig ausfallen.

Antworten zu den offenen Übungen

1. Vergleichen Sie die verschiedenen Kosten, die entstehen, wenn Sie Ihre Server in einer Cloud betreiben, mit den Kosten für den Betrieb vor Ort.

   In einer Cloud zahlen Sie für die CPU-Nutzung und den Datenspeicher für den vom Anbieter gemessenen Zeitraum. Aber Sie haben keine Hardware-Kosten. On-Premises haben Sie fixe Kosten für die Hardware ebenso wie für andere Geräte wie etwa die Klimaanlage sowie laufende Kosten für Strom und Wartung.

2. Sie sind vom Nahen Osten aus tätig, haben aber viele Kunden in Europa und im Fernen Osten. Beschreiben Sie, wo Sie Ihre Dienste im Rahmen eines Cloud-Angebots platzieren würden.

   Nutzen Sie eine Region im Nahen Osten für Ihre eigenen Büros und Kunden im Nahen Osten. Eine Region in Europa ist wichtig, um die General Data Protection Regulation (GDPR) einzuhalten. Möglicherweise benötigen Sie eine Region in China, um das chinesische Gesetz zum Schutz personenbezogener Daten (PIPL) einzuhalten. In jedem Fall ist eine Region im Fernen Osten und in Europa wertvoll für eine bessere Performance bei der Interaktion mit Kunden an diesen Orten.

   Wählen Sie innerhalb jeder Region mehrere Zonen, um sich gegen den Ausfall einer einzelnen Zone zu schützen.

Linux Professional Institute

Thema 052: Open-Source-Softwarelizenzen

052.1 Konzepte von Open-Source-Softwarelizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 052.1

Gewichtung

3

Hauptwissensgebiete

- Verständnis der Definitionen von Open Source Software und freier Software
- Bewusstsein für andere Arten kostenfreier Software
- Bewusstsein für wichtige Ereignisse in der Geschichte von Open Source
- Verständnis, was eine Lizenz ist und welche Rechte Lizenzen gemeinhin regeln
- Verständnis, wie bestehende Software zur Erstellung abgeleiteter Werke verwendet werden kann
- Verständnis der Kompatibilität und Inkompatibilität von Lizenzen
- Verständnis von Doppellizenzierung und bedingter Lizenzierung
- Verständnis der Konsequenzen von Lizenzverstößen
- Verständnis der Prinzipien des Urheberrechts und des Patentrechts und wie diese von Open-Source-Softwarelizenzen beeinflusst werden

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Free Software Foundation (FSF) Definition von freier Software
- Open Source Initiative (OSI) Definition von Open Source Software
- Lizenzen
- Verträge

- Public-Domain-Software
- Freeware
- Shareware
- License Stewards
- Erlaubnis zur Nutzung, Änderung und Verbreitung von Code und Software
- Abgeleitete Werke und Wiederverwendung von Code
- Closed Source / proprietäre Software
- Bezahlter Vertrieb
- Modifizierter und unmodifizierter Softwarevertrieb
- Hosting von Software als kostenpflichtiger Dienst
- Lizenzkompatibilität
- Doppelte und mehrfache Lizenzierung
- Bedingte Lizenzierung
- Software-Patente
- Explizite und implizite Patentlizenzvergabe

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	052 Open-Source-Softwarelizenzen
Lernziel:	052.1 Konzepte von Open-Source-Softwarelizenzen
Lektion:	1 von 1

Einführung

Jeder Softwareentwickler weiß es zu schätzen, wenn ein Problem bereits effizient gelöst wurde. Ist die Lösung online verfügbar, ist der verständliche Reflex, sie auszuprobieren: den vorhandenen Quellcode in die eigene Codebasis kopieren oder verlinken, ihn testen und dort belassen, wenn er funktioniert — und ihn dann vergessen.

Aber es ist Vorsicht geboten. In den meisten Fällen steht der im Internet verfügbare Quellcode unter einer *Lizenz* für freie und Open Source Software (FOSS). Diese Lektion beschreibt einige grundlegende Konzepte von FOSS aus rechtlicher Sicht und erläutert, warum es ratsam ist, FOSS-Quellcode nicht für selbstverständlich zu halten, sondern die Lizenzbedingungen genau zu beachten.

Um die rechtlichen Verpflichtungen im Zusammenhang mit Software zu verstehen, sollten Sie wissen, dass fast jede Software einer Lizenz unterliegt — und eine Lizenz ist eine Art Vertrag. Jede Software, einschließlich Websites, sollte mit einer schriftlichen Version ihrer Lizenz (auch als “Geschäftsbedingungen” bezeichnet) ausgeliefert werden. Bei einigen Programmen und Websites müssen Sie ein Kästchen abhaken, das besagt, dass Sie die Geschäftsbedingungen zur Kenntnis

genommen haben — und das sollten Sie auch, obwohl die meisten Benutzer das nicht tun. In jedem Fall akzeptieren Sie die Lizenz implizit, indem Sie die Software verwenden.

Definitionen von Open Source Software und freier Software

Freie und Open Source Software gibt es schon seit geraumer Zeit, und die Begriffe werden oft zusammen verwendet. Allerdings gibt es gewisse Unterschiede zwischen dem, was manche als "freie Software" bezeichnen, und den Fachbegriffen *freie Software* und *Open Source Software*. Spoiler: *Open Source* bedeutet nicht nur, dass jeder den Quellcode einsehen kann — das wäre eher "quelloffene" Software oder Software, deren Quellcode verfügbar ist. Freie und Open Source Software ist viel mehr als das.

Freie und Open Source Software steht im Gegensatz zu anderen Arten, die als *proprietär* oder *closed source* gelten, weil sie nicht alle in dieser Lektion besprochenen Freiheiten bieten.

Die vielleicht am häufigsten zitierte Definition von freier Software lautet:

> Man denke an "Redefreiheit", nicht an "Freibier".
>
> — Richard Stallman, Selling Free Software

Wir werden diesen anschaulichen Vergleich in den folgenden Abschnitten weiter ausführen.

Frei wie in Redefreiheit: Wahre Freiheit für Benutzer

Entwickler können sich dafür entscheiden, auf die Einnahmen und Schwierigkeiten des Verkaufs ihrer Software zu verzichten, und sie ohne Entschädigung herausgeben. Im Englischen ist diese Software "free" in dem Sinne, dass niemand dafür bezahlen muss, so wie Freibier, das verschenkt wird. Diese kostenlose Bereitstellung sagt allerdings nichts über weitere Lizenzbedingungen aus und lässt Benutzer mit der Verpflichtung zurück, immer wieder genau hinzuschauen: Es könnte zum Beispiel sein, dass nicht die notwendigen Rechte eingeräumt wurden, um die Software zu verändern oder weiterzugeben.

Entwickler können sich jedoch auch dafür entscheiden, ihre Software zu "freier Software" im Sinne Stallmans zu machen. Dieser Begriff (den er in den 1980er Jahren geprägt hat) bezieht sich auf die grundsätzlichen Freiheiten des Benutzers, die wie folgt zusammengefasst werden können:

1. die Software ausführen
2. sie studieren
3. sie an andere weitergeben (*redistribute*)
4. Kopien der geänderten Versionen weitergeben

“Freie Software” nach dieser Definition wird von der Free Software Foundation (FSF) unterstützt, die auch den Begriff *Copyleft* eingeführt hat (der keine juristische Bedeutung hat, sondern für eine philosophische Überlegung steht), um freie Softwarelizenzen zu charakterisieren.

Open Source Software

Aus der Bewegung für freie Software kamen in den späten 1990er Jahren Befürworter von Open Source, um freie Software verständlicher und bei Benutzern außerhalb der Bewegung populärer zu machen. 1998 wurde eine gemeinnützige Stiftung, die *Open Source Initiative* (OSI), gegründet, und Linus Torvalds, der ursprüngliche Autor des Linux-Kernels, unterstützte das Konzept. Die Open Source Initiative formalisierte die *Open Source Definition*, die die folgenden Kriterien umfasst:

1. *Freie Weitergabe*: Man kann frei entscheiden, wie man das Programm weitergibt, ob kostenlos oder zum Verkauf, solange keine Lizenzgebühr verlangt wird. Diese Freiheit schließt die Einbindung des Programms in ein anderes Programm ein.
2. *Verfügbarkeit des Quellcodes*, entweder online oder zusammen mit der Software.
3. Erlaubnis zur Erstellung und Verbreitung *abgeleiteter Werker* und Änderungen.
4. *Integrität des Quellcodes des Autors*: Modifikationen können eingeschränkt werden, wenn es den Empfängern erlaubt ist, das Programm durch Patches zum Zeitpunkt der Erstellung zu modifizieren und diese Patches zusammen mit dem Quellcode zu verteilen. Die Weitergabe von Software, die aus modifiziertem Quellcode *gebaut* wurde, darf nicht eingeschränkt werden.
5. *Keine Diskriminierung von Personen oder Gruppen*: Eine Lizenz, die die Nutzung der Software “nur für Lehrer” erlaubt, würde beispielsweise nicht der Open-Source-Definition entsprechen.
6. *Keine Diskriminierung von Arbeitsbereichen*: Schränken Sie zum Beispiel die kommerzielle Nutzung nicht ein.
7. *Weitergabe der Lizenz*: Jeder, der das Programm erhält, erhält auch dieselbe, ursprüngliche Lizenz.
8. *Die Lizenz darf sich nicht auf ein bestimmtes Produkt beziehen.*
9. *Die Lizenz darf andere Software nicht einschränken*: Andere Software, die mit dieser Software gebündelt ist, könnte zum Beispiel eine andere Lizenz haben.
10. *Die Lizenz muss technologieneutral sein.*

Freiheit vs. Open Source

Die Free Software Foundation lehnt den Begriff “Open Source” ab, da er das Hauptziel der Freiheit verschleiert. Obwohl die Befürworter von freier Software und Open Source Software scheinbar

dasselbe Konzept verfolgen und dafür eintreten, haben die Bewegungen unterschiedliche Beweggründe. Vereinfacht gesagt, betonen die Befürworter von freier Software die Rechte von Entwicklern und Benutzern, während die Befürworter von Open Source Software den weit verbreiteten Einsatz und Erfolg der Software in den Vordergrund stellen.

So gut wie alle freie Software gilt als Open Source, während es viele Lizenzen gibt, die zwar als Open Source gelten (wie von der OSI bestätigt), aber nach Ansicht der FSF nicht frei sind.

Da die Unterschiede zwischen freiem und offenem Quellcode mehr die Ziele und Motivationen als den Inhalt der Lizenzen betreffen und beide Begriffe häufig verwendet werden, beziehen sich viele Befürworter auf beide Definitionen zusammen, indem sie die Ausdrücke *Freie und Open Source Software* (FOSS) oder *Free/Libre and Open Source Software* (FLOSS) verwenden. Der Begriff “libre” verweist in vielen romanischen Sprachen auf die Freiheit.

Andere Arten kostenloser Software

Neben den beiden Kategorien FOSS und proprietäre Software gibt es noch eine Reihe anderer Vertriebsstrategien, beispielsweise folgende:

Shareware

Der Begriff bezieht sich im Allgemeinen auf proprietäre Software, die zwar zum Verkauf steht, aber mit eingeschränkter Funktionalität kostenlos genutzt werden kann, bis sich der Benutzer zum Kauf der Vollversion entschließt.

Freeware

Der Begriff beschreibt Software, die kostenlos und ohne Nutzungsbeschränkung vertrieben wird, aber nicht unbedingt der formalen Definition freier Software entspricht. Freeware ist in vielen Fällen proprietär, und der Quellcode wird oft gar nicht veröffentlicht.

Software mit verfügbarem Quellcode

Manchmal stellen Entwickler proprietärer Software ihren Quellcode zur Verfügung (beispielsweise zur einfacheren Fehlersuche), machen aber den Erwerb einer proprietären Lizenz zur Bedingung für die Verwendung des Quellcodes in anderen Projekten. Diese Art der Verfügbarkeit mit strengen Einschränkungen sollte nicht mit echter Open Source Software verwechselt werden.

Shared Source Software

Der Begriff wurde 2001 von Microsoft eingeführt, als das Unternehmen beschloss, einen Teil seines Software-Quellcodes online zu Forschungs- und Testzwecken zur Verfügung zu stellen. Verwechseln Sie diese enge Definition nicht mit Shareware oder quelloffener Software.

Public Domain (gemeinfreie) Software

Dies ist Software, bei der die Autoren auf das Copyright verzichtet haben. Diese Definition gilt nicht in allen Rechtsordnungen (vor allem nicht in denjenigen, in denen dem Autor das *droit d'auteur*, *author's rights* oder *Urheberrecht* gewährt wird — wie etwa in Frankreich oder Deutschland). Es wurden Lizenzen wie die "Unlicense" eingeführt, die die gleiche Wirkung haben sollen. Außerdem kann Software auch dann gemeinfrei werden, wenn die Dauer des Urheberrechts abgelaufen ist.

Grundsätze des Urheberrechts und die Auswirkungen der Open-Source-Softwarelizenzen

Das Wichtigste zuerst: Wenn für eine bestimmte Quellcodedatei oder ein bestimmtes Projekt keine Lizenzinformationen verfügbar sind, können Sie nicht davon ausgehen, dass die Datei oder das Projekt keinen Urheberrechtsschutz genießt. Tatsächlich ist das Gegenteil der Fall, zumindest seit die meisten Nationen weltweit die *Berner Übereinkunft* von 1887 unterzeichnet haben.

In diesem Vertrag haben sich die Unterzeichnerstaaten darauf geeinigt, dass ein literarisches oder künstlerisches Werk urheberrechtlich geschützt ist, sobald es existiert (oder anders gesagt, sobald es in einem Medium "fixiert" ist). Das bedeutet, dass ein Urheber das Urheberrecht nicht registrieren oder beantragen muss. In einigen Ländern kann er dies dennoch tun, und in einigen Gerichtsbarkeiten, einschließlich der USA, kann eine Registrierung für Klagen wegen Rechtsverletzung erforderlich sein.

Außerdem verpflichten sich die Unterzeichner, die Urheberrechte aller Autoren eines anderen Unterzeichnerlandes zu respektieren, was im November 2022 181 der 195 Länder der Welt betraf.

Allerdings ist nicht alles, was jemals geschaffen wurde, urheberrechtlich geschützt. Um als urheberrechtlich geschütztes *Werk* zu gelten, muss die Schöpfung einige grundlegende Kriterien erfüllen: Fakten und Ideen können beispielsweise nicht urheberrechtlich geschützt werden, aber ein Text, der eine Idee erläutert, kann geschützt werden, wenn er ein bestimmtes Maß an *Originalität* erreicht. Viele Gerichtsbarkeiten prüfen derzeit, wie viel künstliche Intelligenz eingesetzt werden darf, um ein Werk zu schaffen, das Originalität und damit urheberrechtlichen Schutz beanspruchen kann.

Je nach Rechtsprechung sind Computerprogramme wie literarische Werke urheberrechtlich geschützt: Das heißt, das Urheberrecht gilt nicht für die Idee oder den Algorithmus, sondern für deren Umsetzung im Quellcode.

Das Urheberrecht gibt dem Urheber das ausschließliche Recht (unter anderem), das Werk zu vervielfältigen, zu verändern, Unterlizenzen zu vergeben, zu verbreiten und zu veröffentlichen. Der Empfang des Werks ist frei, man braucht also keine Lizenz, um ein Buch zu lesen oder ein

Lied im Radio zu hören, solange man dafür keine dauerhafte Kopie anfertigen muss.

Da die Rechte des Urhebers ohne Eintragung entstehen, muss jeder, der das Werk eines anderen Urhebers vervielfältigen, verändern, unterlizenzieren, verbreiten oder veröffentlichen will, zunächst die entsprechende Erlaubnis einholen. Hier kommen Lizenzen ins Spiel, als Verträge zwischen dem Urheber eines Werks und einer Person, die einige der ausschließlichen Rechte des Urhebers ausüben möchte.

FOSS-Lizenzen werden für jedermann kostenlos angeboten. Jeder kann seine eigene Lizenz erstellen und versuchen, sie von der OSI oder der FSF genehmigen zu lassen. Es wird jedoch dringend empfohlen, eine bestehende FOSS-Lizenz zu verwenden, da sie allgemein akzeptiert wird und sachkundige Softwareanwender mit dem Inhalt der Lizenz und ihren Verpflichtungen vertraut sind. Tatsächlich sind nur eine Handvoll der vielen von der FSF oder OSI genehmigten Lizenzen im allgemeinen Gebrauch.

Grundsätze des Patentrechts

Im Gegensatz zum Urheberrecht, das keine Ideen schützt, schützen Patente Erfindungen (Ideen), ohne dass die Idee (bisher) in einer Maschine oder einem Verfahren fixiert sein muss. Ein weiterer Unterschied besteht darin, dass Erfinder Patente ausdrücklich anmelden und beim Patentamt des Landes registrieren lassen müssen, in dem der Schutz beantragt wird.

Ohne zu sehr in das Patentrecht und seine Anforderungen einzutauchen, wollen wir nur auf einen ganz zentralen Punkt hinweisen: Der Patentschutz setzt (neben anderen Kriterien) eine Idee mit einem bestimmten technischen Bezug voraus. Historisch gesehen sind die Ideen für ein neues Kochrezept oder ein neues Brettspiel nicht patentfähig, während die Ideen für eine neue Kochmaschine oder eine Spielkonsole patentfähig sein können.

Im Zusammenhang mit der Programmierung von Computern stellt sich die Frage, ob Software patentrechtlich geschützt werden kann. Dies hängt sowohl von der Rechtsprechung als auch von der jeweiligen Anwendung ab: In Deutschland beispielsweise sind Computerprogramme als solche generell vom Patentschutz ausgeschlossen. Wenn jedoch Programme mit einem physischen Gegenstand kombiniert werden — wenn beispielsweise Software ein automatisches Bremssystem in einem Auto steuert — kann unter Umständen Patentschutz für die Anwendung beantragt werden, die das physische Element der Bremse umfasst, und nicht die Software als solche.

In anderen Gerichtsbarkeiten, etwa in den USA, können Patente für Computerprogramme als solche erteilt werden, je nach Entwicklung der Rechtsprechung.

Das Konzept des Patentschutzes sollte bei der Lizenzierung von FOSS berücksichtigt werden. Einige Lizenzen (wie die GPLv3) erlauben Patente auf FOSS-Software, indem sie die Nutzung der

Software explizit erlauben. Einige Lizenzen erwähnen Patente jedoch nicht einmal (wie die BSD-3-Clause-Lizenz), und andere Lizenzen schließen sie explizit aus (wie die Creative-Commons-Lizenzen). Unter bestimmten Umständen können Patente jedoch durch die Lizenz impliziert oder in die Lizenz hineingelesen werden.

NOTE Die verschiedenen FOSS-Lizenzen werden in den folgenden Lektionen behandelt.

Vor allem bei eingebetteter (embedded) Software, wie beispielsweise Software in Audiogeräten, sollte ein besonderes Augenmerk auf Patente im Zusammenhang mit der Software gelegt werden, etwa durch Patentprüfungen bei den Autoren der Software.

Lizenzverträge

Wie bereits erwähnt, sind Lizenzen Verträge zwischen dem Autor eines Werkes und jemandem, der einige der ausschließlichen Rechte des Autors ausüben möchte. Einige der umfangreicheren FOSS-Lizenzen, wie die GNU General Public Licenses Version 2 und Version 3, enthalten Bestimmungen zur Vertragskündigung. FOSS-Lizenzen enthalten immer Rechteeinräumungen bezüglich der zentralen Freiheiten. Üblicherweise werden die folgenden Rechte angegeben oder können in den Lizenztext gelesen werden:

> ...das Recht, die Software zu verwenden, zu kopieren, zu ändern, zusammenzuführen, zu veröffentlichen, weiterzugeben, Unterlizenzen zu vergeben und/oder zu verkaufen...
>
> — MIT-Lizenz

License Stewards (Lizenzverwalter) sind Personen oder, in vielen Fällen, Organisationen wie die FSF, die die Versionen der Lizenzen verwalten; so erklärt beispielsweise Abschnitt 9 der GPLv2 die FSF zum License Steward:

> Die Free Software Foundation kann von Zeit zu Zeit überarbeitete und/oder neue Versionen der General Public License veröffentlichen. Solche neuen Versionen werden im Geiste der aktuellen Version ähnlich sein, können sich aber im Detail unterscheiden, um neue Probleme oder Anliegen zu adressieren.
>
> — GNU General Public License Version 2

Einige License Stewards veröffentlichen auch häufig gestellte Fragen (FAQs), die bei der Beantwortung von Fragen zu den Lizenzen helfen und als Gesprächsgrundlage zwischen Lizenznehmer und Lizenzgeber dienen können.

Verbreitung

Die Verbreitung oder Weitergabe (engl. “distribution”) von Software (insbesondere in Form von Binär- oder Quellcode) ist ein zentraler Aspekt der meisten FOSS-Lizenzen, da die bloße *Nutzung* von FOSS-Software normalerweise keine Lizenzverpflichtungen auslöst.

> Andere Handlungen als Vervielfältigung, Verbreitung und Bearbeitung werden von dieser Lizenz nicht berührt; sie fallen nicht in ihren Anwendungsbereich. Der Vorgang der Ausführung des Programms wird nicht eingeschränkt…
>
> — GNU General Public License Version 2

Die meisten Lizenzverpflichtungen entstehen erst mit der Weitergabe, also der Weitergabe einer veränderten oder unveränderten Kopie, sei es auf einem materiellen Medium wie einer CD oder per Download. In einigen Fällen werden die Lizenzbedingungen auch ausgelöst, wenn die Software auf einem Server läuft (also ohne dass der Quellcode jemals weitergegeben wird), während der Benutzer mit der Software interagiert.

Die Weitergabe von ausführbarer Software kann je nach Art der FOSS-Lizenz auch die Übergabe des Quellcodes, des Lizenztextes und der Copyright-Hinweise erfordern. Einige Lizenzen verlangen, dass Änderungshinweise in den Quellcode eingefügt werden, wenn modifizierter Code weitergegeben wird.

Die Verbreitung könnte auch Copyleft-Effekte auslösen: Bei der Verbreitung von GPLv3-lizenzierter Software, die verändert wurde, muss beispielsweise der Quellcode der gesamten veränderten Software möglicherweise unter der GPLv3 freigegeben werden.

FOSS darf zwar verkauft werden, aber in der Regel dürfen keine Lizenzgebühren erhoben werden. Das bedeutet, dass jemand beispielsweise Software unter der GPLv3-Lizenz nur als Binärdatei verkaufen kann, aber da der Quellcode verfügbar ist (oder angeboten werden muss), können Interessierte jederzeit entscheiden, ob sie die Quellen (vielleicht von jemand anderem kostenlos) beziehen und die Software aus den Quellen erstellen.

Abgeleitete Werke

Wenn eine Gruppe von Entwicklern Code aus einem anderen Projekt in ihr eigenes Projekt übernimmt, kann das Ergebnis ein *abgeleitetes Werk* (*derivative work*) sein. Die Details variieren von einem Projekt zum anderen und von einer Lizenz zur anderen, und da die Details in den technischen Bereich der Softwareentwicklung fallen, sagen wir hier einfach, dass die Entwickler sicherstellen müssen, dass sie den Code in einer Weise integrieren, die mit der Lizenz konform ist.

Der wichtigste Punkt in Bezug auf abgeleitete Werke ist, dass bei einigen Lizenzen, wie der GPL,

ein abgeleitetes Werk unter derselben Lizenz veröffentlicht werden muss. Solche Lizenzen werden *reziprok* (also “wechselseitig”) genannt.

Die unmittelbare praktische Bedeutung der Forderung nach Wechselseitigkeit besteht darin, dass Sie, wenn Sie GPL-Code in Ihrem eigenen Projekt in einer Weise verwenden, die Ihren Code zu einem abgeleiteten Werk macht, Ihren Quellcode offenlegen und anderen erlauben müssen, Produkte darauf aufzubauen. Diese Lizenzierungsanforderung ist von Entwicklern explizit erwünscht, die mehr Menschen dazu ermutigen wollen, freie Lizenzen zu verwenden. Allerdings verringert die Lizenz die Attraktivität der GPL für einige Entwickler, die möglicherweise freien Code verwenden könnten.

Viele andere freie und Open-Source-Lizenzen stellen diese Anforderung nicht, sie werden als *permissive* (also “freizügige”) Lizenzen bezeichnet.

Folgen von Lizenzverstößen

Wenn zum Zeitpunkt der Verbreitung die Lizenzbedingungen nicht erfüllt sind (wenn beispielsweise GPLv3-lizenzierte Software als Binärdatei verbreitet wird, ohne dass auch der Quellcode beiliegt), entspricht das einer Verletzung des Lizenzvertrags. Die Folgen von Lizenzverletzungen hängen von der jeweiligen Lizenz ab. Eine Verletzung der GPLv3-Lizenz kann etwa zur Beendigung der Lizenz führen. Alle weiteren Handlungen, die die Zustimmung des Autors erfordern, zum Beispiel die Verbreitung der Software, stellen dann Urheberrechtsverletzungen dar.

Wenn ein Unternehmen GPLv3-lizenzierte Software in ein Produkt einbaut und gegen die Lizenz verstößt, kann Klage eingereicht werden, die das Unternehmen letztlich zum Rückruf seiner Produkte verpflichtet. Vorsätzliche Urheberrechtsverletzungen können sogar strafrechtlich verfolgt werden.

Einige Lizenzen enthalten spezielle Klauseln, die es dem Lizenznehmer erlauben, den Verstoß innerhalb von 30 Tagen nach der Benachrichtigung zu beheben. Gelingt es der Person, die die Software vertreibt, innerhalb dieser Frist die Lizenz einzuhalten, ist die Lizenz wiederhergestellt.

Lizenz-Kompatibilität und -Inkompatibilität

Große Softwareprojekte enthalten oft Software unter verschiedenen Lizenzen, wobei jede Lizenz ihre eigenen Anforderungen stellt. Solche Projekte können auf Hindernisse stoßen, wenn sie Software verwenden, deren Lizenz für abgeleitete Werke verwendet werden muss. Dies liegt daran, dass sich die Lizenzbedingungen solcher Copyleft-Lizenzen unterscheiden können, was sie zueinander inkompatibel macht. Die Veröffentlichung oder Verbreitung eines Softwareprojekts, das Komponenten unter verschiedenen Copyleft-Lizenzen integriert, ist eventuell nicht möglich,

ohne eine der Lizenzen zu verletzen.

Einige Copyleft-Lizenzen führen ausdrücklich kompatible Lizenzen auf, so dass es einfacher ist, eine Copyleft-lizenzierte Komponente unter einer anderen Copyleft-Lizenz zu verwenden.

Die meisten permissiven Lizenzen sind mit anderen Lizenzen kompatibel. Zum Beispiel können MIT-lizenzierte Komponenten in einem GPLv3-lizenzierten Projekt verwendet werden, ohne Lizenzverletzungen zu riskieren. Allerdings kann eine GPLv3-lizenzierte Komponente nicht in jedem Fall in einem MIT-lizenzierten Projekt verwendet werden, ohne die Bedingungen der GPLv3 zu verletzen. Kompatibilität ist also nicht immer in beide Richtungen gegeben.

Bevor ein Softwareprojekt auf den Markt gebracht wird, sollten Entwickler und ihre Rechtsberater eine gründliche Prüfung der Lizenzkompatibilität durchführen, um Lizenzverstöße zu vermeiden. Open Source Compliance Management sollte in die frühen Phasen eines Softwareentwicklungsprozesses integriert werden, um Verzögerungen bei der Auslieferung der Software zu vermeiden, die beispielsweise durch Lizenzinkompatibilität entstehen. Durchsuchen Sie den Quellcode gründlich nach geltenden Lizenzen (beispielsweise mithilfe von Software Scan Tools) und prüfen Sie, ob alle Lizenzbedingungen erfüllt sind.

Doppellizenzierung und Mehrfachlizenzierung

Manche Software kann unter mehreren Lizenzen verfügbar sein. Ein Lizenzgeber kann sich beispielsweise dafür entscheiden, sein Projekt sowohl unter einer Copyleft-Lizenz wie der GPLv3 als auch unter einer proprietären Lizenz zu lizenzieren. Die proprietäre Lizenz kann erforderlich sein, wenn der potenzielle Lizenznehmer den Code in sein eigenes proprietäres Produkt integriert. Jeder Entwickler entscheidet, ob er sich an die Bedingungen der GPLv3 halten kann oder die proprietäre Lizenz erwerben muss, was höchstwahrscheinlich mit einer Lizenzgebühr verbunden ist.

Wie bereits erwähnt, kann manche Software Komponenten enthalten, die unter verschiedenen Lizenzen mit unterschiedlichen Lizenzbedingungen stehen. Obwohl die meisten Lizenzen nicht zu Inkompatibilität führen, können verschiedene Lizenzen unterschiedliche Anforderungen für verschiedene Teile der Software mit sich bringen.

Geführte Übungen

1. Wofür steht das Akronym FOSS?

2. Welche der folgenden Punkte gehören explizit zu den wesentlichen Freiheiten von Benutzern freier Software?

Software ausführen	
Software studieren	
Software kopieren	
Software ändern	
Software veröffentlichen	
Software verbreiten	

3. Darf Quellcode, der im Internet ohne Lizenzinformationen gefunden wird, von jedermann verändert und verbreitet werden? Bitte erläutern Sie.

4. Kann Software patentiert werden?

Ja	
Nein	
Es kommt darauf an	

5. Was ist ein License Steward?

Dasselbe wie ein Lizenzgeber	
Jemand, der zukünftige Versionen einer Lizenz vorschlagen kann	
Jemand, der eine Lizenz beenden kann	
Jemand, der Software von Flugzeugen verwaltet	

6. Ist Lizenzkompatibilität immer in beide Richtungen gegeben?

Ja, wenn zwei Lizenzen kompatibel sind, spielt es keine Rolle, ob A in B enthalten ist oder B in A enthalten ist.	
Nein, in manchen Fällen kann Lizenz A in ein Projekt unter Lizenz B eingebunden werden, aber B darf nicht unter Lizenz A weitergegeben werden.	
Nein, Lizenzkompatibilität ist immer unidirektional.	

Offene Übungen

1. Sind die GPLv2 und die LGPLv2.1 kompatibel? Bitte erläutern Sie kurz.

2. Kann Software unter Open-Content-Lizenzen (wie CC-BY) veröffentlicht werden?

Ja, CC-Lizenzen können auf jedes urheberrechtsfähige Werk angewendet werden.	
Nein, Software kann nur durch Patente geschützt werden.	
Nein, CC-Lizenzen gelten nicht für Software.	

3. Warum ist die Verbreitung im Zusammenhang mit der Lizenzierung von FOSS wichtig?

Zusammenfassung

Diese Lektion bietet eine Einführung in die grundlegenden Konzepte von freier und Open Source Software sowie in die zugrundeliegenden Konzepte von Urheberrecht und Patenten. Sie erklärt einige der Grundlagen und die Geschichte von freier Software und Open Source Software und hilft, beide von proprietären Lizenzkonzepten zu unterscheiden. Die Open Source Definition der OSI hilft, Lizenzen als Open-Source-Lizenzen zu kategorisieren.

Obwohl es bereits Dutzende von FOSS-Lizenzen gibt, steht es jedem frei, weitere Lizenzen einzuführen.

Das Konzept der Lizenzierung ist nicht zu unterschätzen: Lizenzen geben die Erlaubnis, mit der Software auf eine Art und Weise umzugehen, die sonst ausschließlich dem Autor vorbehalten ist. Verstöße gegen Lizenzen können zu Rechtsstreitigkeiten führen. Daher sollte man sich rechtlich beraten lassen, bevor man Software weitergibt oder in anderen Projektcode integriert.

Antworten zu den geführten Übungen

1. Wofür steht das Akronym FOSS?

   Free and Open Source Software

2. Welche der folgenden Punkte gehören explizit zu den wesentlichen Freiheiten von Benutzern freier Software?

Software ausführen	X
Software studieren	X
Software kopieren	
Software ändern	X
Software veröffentlichen	
Software verbreiten	X

3. Darf Quellcode, der im Internet ohne Lizenzinformationen gefunden wird, von jedermann verändert und verbreitet werden? Bitte erläutern Sie.

   Nein. Quellcode ist, wenn er die Schwelle der Originalität erreicht, unmittelbar als literarisches Werk urheberrechtlich geschützt. Da Änderung und Verbreitung ausschließliche Rechte des Urhebers darstellen, darf niemand außer dem Urheber selbst dies tun oder die Erlaubnis dazu erteilen.

4. Kann Software patentiert werden?

Ja	
Nein	
Es kommt darauf an	X

5. Was ist ein License Steward?

Dasselbe wie ein Lizenzgeber	
Jemand, der zukünftige Versionen einer Lizenz vorschlagen kann	X
Jemand, der eine Lizenz beenden kann	

Jemand, der Software von Flugzeugen verwaltet	

6. Ist Lizenzkompatibilität immer in beide Richtungen gegeben?

Ja, wenn zwei Lizenzen kompatibel sind, spielt es keine Rolle, ob A in B enthalten ist oder B in A enthalten ist.	
Nein, in manchen Fällen kann Lizenz A in ein Projekt unter Lizenz B eingebunden werden, aber B darf nicht unter Lizenz A weitergegeben werden.	X
Nein, Lizenzkompatibilität ist immer unidirektional.	

Antworten zu den offenen Übungen

1. Sind die GPLv2 und die LGPLv2.1 kompatibel? Bitte erläutern Sie kurz.

   Wenn Software A unter der GPLv2 und Software B unter der LGPLv2.1 lizenziert ist, ist es möglich, B in A zu verwenden, da die LGPLv2.1 die Verwendung unter den Bedingungen der GPLv2-Lizenz erlaubt. Allerdings kann A nicht in B unter den Bedingungen der LGPLv2.1 verwendet werden. Wenn A in B verwendet wird, muss die gesamte Software unter der GPLv2 lizenziert werden, was möglich ist, da die LGPLv2.1 die Verwendung der Software unter der GPLv2.0 erlaubt.

2. Kann Software unter Open-Content-Lizenzen (wie CC-BY) veröffentlicht werden?

Ja, CC-Lizenzen können auf jedes urheberrechtsfähige Werk angewendet werden.	X
Nein, Software kann nur durch Patente geschützt werden.	
Nein, CC-Lizenzen gelten nicht für Software.	

3. Warum ist die Verbreitung im Zusammenhang mit der Lizenzierung von FOSS wichtig?

   Viele FOSS-Lizenzverpflichtungen werden mit der Verbreitung der Software ausgelöst. Wenn Software nie weitergegeben, sondern nur verändert und in einem geschlossenen System (zum Beispiel in einer Abteilung eines Unternehmens) verwendet wird, kann es möglich sein, die Software zu nutzen, ohne die Lizenzverpflichtungen einzuhalten.

052.2 Copyleft-Softwarelizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 052.2

Gewichtung

3

Hauptwissensgebiete

- Verständnis des Konzepts der Copyleft-Softwarelizenzen
- Verständnis der Rechte, die durch Copyleft-Softwarelizenzen gewährt werden
- Verständnis der Verpflichtungen, die durch Copyleft-Softwarelizenzen entstehen
- Verständnis der Haupteigenschaften gängiger Copyleft-Softwarelizenzen
- Verständnis der Kompatibilität von Copyleft-Softwarelizenzen mit anderen Softwarelizenzen
- Kenntnis der Begriffe “reziproke Lizenz” und “restriktive Lizenz”

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Copyleft
- Verteilen (Distributing)
- Weitergabe (Conveying)
- GNU General Public License, version 2.0 (GPLv2)
- GNU General Public License, version 3.0 (GPLv3)
- GNU Lesser General Public License, Version 2 (LGPLv2)
- GNU Lesser General Public License, Version 3 (LGPLv3)
- GNU Affero General Public License, Version 3 (AGPLv3)

- Eclipse Public License (EPL), version 1.0
- Eclipse Public License (EPL), version 2.0
- Mozilla Public Licence (MPL)

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	052 Open-Source-Softwarelizenzen
Lernziel:	052.2 Copyleft-Softwarelizenzen
Lektion:	1 von 1

Einführung

Die Bedeutung von Lizenzen sowohl für die Nutzung, aber auch für die Entwicklung von Software wurde bereits beschrieben. Es wundert daher nicht, dass sich freie Software von Anfang an auch durch neue Ansätze der Lizenzierung auszeichnet: Bedingungen für die uneingeschränkte Nutzung oder die kollaborative Entwicklung von Software müssen juristisch definiert sein, um geschützt und durchgesetzt werden zu können.

Im Bewusstsein, das weltweit in nahezu allen Rechtssystemen fest verankerte Urheberrecht nicht infrage stellen oder gar ersetzen zu können, entstand bereits in den 1980er Jahren ein Ansatz, der die urheberrechtlichen Vorschriften zwar respektiert, aber um neue, vor allem das Prinzip der "Freiheit" betonende Vorschriften ergänzt: *Copyleft*.

Copyleft und die GNU General Public License (GPL)

Richard Stallman, damals Entwickler am renommierten MIT, hatte 1983 das *GNU Project* zur Entwicklung eines nach seinen Vorstellungen "freien" Betriebssystems gegründet. Schnell wurde deutlich, dass der vom Projekt entwickelte Code juristisch abgesichert werden musste, um beispielsweise nicht einfach von kommerziellen Anbietern übernommen und damit "unfrei" zu

werden.

Stallman gründete darum 1985 die gemeinnützige *Free Software Foundation* (FSF), die ihren Auftrag auf ihrer Website wie folgt zusamenfasst: “Die Free Software Foundation setzt sich für die Freiheit der Computerbenutzer ein, indem sie die Entwicklung und Verwendung freier (wie in Freiheit) Software und Dokumentation fördert [...].”

Ein entscheidendes Instrument dafür ist eine Lizenz, die einerseits geltendes Recht (insbesondere das Urheberrecht) respektiert, andererseits die eigenen Vorstellungen von Freiheit juristisch sauber umsetzt. Das Ergebnis war die erste Version der *GNU General Public License* (GPLv1) im Jahr 1989. Diese Lizenz und zahlreiche Artikel — wie etwa der 1992 von Stallman verfasste “What is Free Software?” — machen deutlich, um was es den sich nun auch als “Bewegung” verstehenden Entwicklern freier Software geht.

Den programmatischen Kern bilden bis heute die von Stallman in dem genannten Artikel formulierten “vier essentiellen Freiheiten”, deren Zählung mit “0” beginnt:

> - Die Freiheit, das Programm auszuführen wie man möchte, für jeden Zweck (Freiheit 0)
> - Die Freiheit, die Funktionsweise des Programms zu untersuchen und eigenen Datenverarbeitungbedürfnissen anzupassen (Freiheit 1). Der Zugang zum Quellcode ist dafür Voraussetzung.
> - Die Freiheit, das Programm zu redistribuieren und damit Mitmenschen zu helfen (Freiheit 2).
> - Die Freiheit, das Programm zu verbessern und diese Verbesserungen der Öffentlichkeit freizugeben, damit die gesamte Gesellschaft davon profitiert (Freiheit 3). Der Zugang zum Quellcode ist dafür Voraussetzung.
>
> — Richard Stallman, What is Free Software? (deutsche Übersetzung auf gnu.org)

Anders als Lizenzen kommerzieller Produkte, die *Beschränkungen* hinsichtlich der Nutzung in den Vordergrund stellen, geht es bei freier Software um ein Maximum an Freiheit für Benutzer und Entwickler.

Die Präambel der GNU GPLv1 fasst das wie folgt zusammen:

> Konkret soll die General Public License sicherstellen, dass Sie die Freiheit haben, Kopien freier Software zu verschenken oder zu verkaufen, dass Sie den Quellcode erhalten oder erhalten können, wenn Sie ihn benötigen, dass Sie die Software verändern oder Teile davon in neuen freien Programmen verwenden können; und dass Sie wissen, dass Sie diese Dinge tun dürfen.

> — GNU General Public License, Version 1 (inoffizielle deutsche Übersetzung)

Jeder hat also das Recht, Software unter der GPL uneingeschränkt zu nutzen, weiterzugeben, zu verändern (was dadurch möglich ist, dass der Source Code zugänglich, also “offen” ist) und wiederum die Änderungen weiterzugeben. Es ist sogar möglich, für die Weitergabe der Software Geld zu verlangen:

> Eigentlich ermutigen wir Menschen, die freie Software weitergeben, so viel Geld zu verlangen, wie sie wollen oder können. Wenn eine Lizenz den Benutzern nicht erlaubt, Kopien anzufertigen und diese zu verkaufen, handelt es sich um eine unfreie Lizenz. [...] Freie Programme werden manchmal kostenlos und manchmal für einen beträchtlichen Preis weitergegeben. Oft ist ein und dasselbe Programm auf beide Arten an verschiedenen Stellen erhältlich. Das Programm ist unabhängig vom Preis frei, weil die Benutzer die Freiheit haben, es zu benutzen.
>
> — Free Software Foundation, Selling Free Software

Wenn aber die Freiheiten so weitreichend sind — inwieweit ist Software unter dieser Lizenz geschützt, beispielsweise vor Übernahme in proprietäre Produkte?

Dies ist die Aufgabe des bereits erwähnten Copyleft-Prinzips, das die GPL bereits in Version 1 anwendet, auch wenn der Begriff noch nicht explizit erscheint:

> Sie dürfen das Programm nicht kopieren, modifizieren, unterlizenzieren, vertreiben oder übertragen, außer wie ausdrücklich unter dieser General Public License festgelegt.
>
> — GNU General Public License Version 1 (inoffizielle deutsche Übersetzung)

Das heißt: All diese Freiheiten sind an die Bedingung geknüpft, dass Benutzer bei allem, was sie mit der Software machen, diese Freiheiten bewahren.

Das Copyleft garantiert also nicht nur Freiheiten, sondern fordert von allen Benutzern, dass sie diese Freiheiten auch anderen gewähren. Das wird dadurch erreicht, dass Software unter einer Copyleft-Lizenz (wie etwa der frühen GPLv1) nur dann verändert und weitergegeben werden darf, wenn man die Veränderungen wiederum unter denselben Bedingungen, also unter derselben Lizenz, veröffentlicht.

Das Ideal freier Software, nämlich die kollektive Nutzung und Weiterentwicklung, steht über den persönlichen Bedürfnissen, die einzelne gegebenenfalls in Bezug auf die Software haben. Entscheidend ist der Grundsatz der Gegenseitigkeit: Wer Freiheiten nutzt, muss diese auch gewähren. Man bezeichnet Copyleft-Lizenzen daher oft auch als *reziprok*.

Dieser völlig neue Ansatz einer Softwarelizenz erwies sich bereits in Version 1 der GPL als

juristisch solide und praxistauglich, und so hat die GPL in den vergangenen fast 40 Jahren, in denen sich der moderne IT-Markt überhaupt erst entwickelte, nur zwei größere Überarbeitungen erfahren.

GPLv2 und GPLv3

1991 legte die Free Software Foundation Version 2 der GNU General Public License (GPLv2) vor, die sich über viele Jahre als beliebteste Lizenz freier Softwareprojekte etablierte. So steht bis heute beispielsweise der Kern des Betriebssystems Linux unter der GPLv2.

Im Vergleich zu Version 1 geht es bei Version 2 vor allem um exaktere Begriffsbestimmungen zur Vermeidung von Mehrdeutigkeiten. So erläutert Version 2 deutlich ausführlicher, was unter “source code” zu verstehen ist.

Interessant ist darüber hinaus der neue Abschnitt 7, der das Prinzip der Freiheit und damit die Gültigkeit der Lizenz absolut setzt und keine Kompromisse — beispielsweise die Integration weniger freier Teile in die Software — zulässt:

> Wenn es Ihnen nicht möglich ist, das Programm unter gleichzeitiger Beachtung der Bedingungen in dieser Lizenz und anderer entsprechender Verpflichtungen zu verbreiten, dann dürfen Sie als Folge das Programm überhaupt nicht verbreiten. Wenn zum Beispiel ein Patent nicht die gebührenfreie Weiterverbreitung des Programms durch diejenigen erlaubt, die das Programm direkt oder indirekt von Ihnen erhalten haben, dann besteht der einzige Weg, sowohl das Patentrecht als auch diese Lizenz zu befolgen, darin, ganz auf die Verbreitung des Programms zu verzichten.
>
> — GNU General Public License Version 2 (inoffizielle deutsche Übersetzung)

Erst 16 Jahre später, im Jahr 2007, veröffentlicht die FSF eine neue Version der GPL, um einerseits technischen Neuerungen — etwa der Bereitstellung von Software Services über das Internet — wie auch Fragen der Kompatibilität mit inzwischen auch anderen FOSS-Lizenzen Rechnung zu tragen. Sie bleibt in Bezug auf ihre Kernaussagen jedoch stabil und ergänzt lediglich Details zur weiteren Präzisierung. Schauen wir uns einige dieser Ergänzungen etwas näher an.

Nennt die GPLv2 die Bereitstellung von Software noch allgemein *distribution* (also “Verteilung” oder “Vertrieb”), so präzisiert die GPLv3 diesen Prozess mit zwei neuen Begriffen: *propagation* (also “Verbreitung” oder “Propagierung”) und *conveying* (also “Übertragung” oder “Übermittlung”). Der Grund liegt vor allem darin, dass der Begriff “distribution” weltweit in zahlreichen Gesetzen zum Urheberrecht definiert ist. Um Unklarheiten oder gar Konflikte zu vermeiden, wählt die GPLv3 diese neuen Termini und definiert sie wie folgt:

> Ein Werk zu “propagieren” bezeichnet jedwede Handlung mit dem Werk, für die man, wenn

> unerlaubt begangen, wegen Verletzung anwendbaren Urheberrechts direkt oder indirekt zur Verantwortung gezogen würde, ausgenommen das Ausführen auf einem Computer oder das Modifizieren einer privaten Kopie. Unter das Propagieren eines Werks fallen Kopieren, Weitergeben (mit oder ohne Modifikationen), öffentliches Zugänglichmachen und in manchen Staaten noch weitere Tätigkeiten.
>
> Ein Werk zu "übertragen" bezeichnet jede Art von Propagation, die es Dritten ermöglicht, das Werk zu kopieren oder Kopien zu erhalten. Reine Interaktion mit einem Benutzer über ein Computer-Netzwerk ohne Übergabe einer Kopie ist keine Übertragung.
>
> — GNU General Public License Version 3 (inoffizielle deutsche Übersetzung)

Mit der deutlich wachsenden Zahl kommerzieller Softwareprodukte, deren Verbreitung durch technische Maßnahmen wie Registrierungscodes oder Hardware-Komponenten (sogenannte *Dongles*) von den Herstellern eingeschränkt wird, gibt es in den späten 1990er Jahren international einige juristische Vorstöße, die die Umgehung dieser Maßnahmen unter Strafe stellen. Dieses sogenannte *Digital Rights Management* (DRM), von Gegnern abfällig auch als *Digital Restrictions Management* bezeichnet, ist der FSF ein Dorn im Auge, widerspricht es doch fundamental der Forderung nach der freien Weitergabe von Software.

Als Reaktion enthält Version 3 der GPL einen Passus, nach dem Software unter der GPL nicht unter Berufung auf gesetzliche Vorgaben zum DRM verändert werden darf. Das bedeutet zugleich, dass eine unter der GPLv3 lizenzierte Software zwar DRM nutzen darf, dass anderen aber auch erlaubt ist, solche Maßnahmen zu umgehen.

Häufig fällt in diesem Zusammenhang auch der Begriff *Tivoisierung*, der in ersten Entwürfen der GPLv3 noch explizit auftaucht, dann aber nicht in die endgültige Fassung übernommen wird. Das Wort geht auf die Firma TiVo zurück, die in ihrem digitalen Videorekorder zwar GPLv2-lizenzierte Software nutzte, zugleich aber technisch verhinderte, dass veränderte Software auf das Gerät aufgespielt und genutzt werden konnte. Nach Auffassung der FSF widersprach dies den Grundsätzen der GPL, und nach einigen Diskussionen trägt die GPLv3 dem mit einem Absatz zu sogenannten *user products* Rechnung. Sie legt darin allgemein fest, dass Produkte, die GPLv3-lizenzierte Software nutzen, auch Informationen bereitstellen müssen, wie diese Software verändert werden kann.

Eine weitere Ergänzung betrifft *Patente*, die die FSF auf Software mit dem Hinweis auf deren Behinderung von Freiheit und Innovation grundsätzlich ablehnt. So heißt es schon in der Präambel der GPLv3:

> Schließlich und endlich ist jedes Computerprogramm permanent durch Softwarepatente bedroht. Staaten sollten es nicht zulassen, dass Patente die Entwicklung und Anwendung von Software für allgemein einsetzbare Computer einschränken, aber in Staaten, wo dies

> geschieht, wollen wir die spezielle Gefahr vermeiden, dass Patente dazu verwendet werden, ein freies Programm im Endeffekt proprietär zu machen. Um dies zu verhindern, stellt die GPL sicher, dass Patente nicht verwendet werden können, um das Programm nicht-frei zu machen.
>
> — GNU General Public License Version 3 (inoffizielle deutsche Übersetzung)

Es folgen im Lizenztext noch einige Passagen, die sowohl die Einbindung von Code unter einem Patent durch eine "nicht-exklusive, weltweite, gebührenfreie Patentlizenz" des Lizenzgebers ermöglichen, um Benutzer solchen Codes vor Streitigkeiten zwischen Patentinhabern und Lizenznehmern zu schützen.

Die GNU Affero General Public License (AGPL)

Mit der zunehmenden Verfügbarkeit und Leistungsfähigkeit des Internet entstehen immer mehr Angebote, bei denen die Software lediglich auf den Servern von Anbietern (*Application Service Provider*, ASP) installiert ist und deren Dienste von Kunden über das Internet abgerufen werden. Dieser Trend hat den Namen *Software as a Service* (SaaS).

Hier bot die GPLv2 keine Klarheit, ob und wie in solchen Fällen die Bereitstellung des (möglicherweise vom Anbieter veränderten) Source Codes zu erfolgen habe. Version 3 der GPL schließt diese als *ASP loophole* bekannte Lücke, indem sie in Abschnitt 13 ausdrücklich auf eine weitere Lizenz der FSF aus dem Jahr 2007 verweist — die *GNU Affero General Public License* Version 3 (GNU AGPLv3). Der Name geht auf die Firma Affero zurück, die noch die ersten beiden Versionen dieser Lizenz entwickelte und veröffentlichte.

Die AGPLv3 entspricht im Grunde der GPLv3, regelt aber im Abschnitt "Remote Network Interaction" explizit das ASP-Problem. Überdies enthalten beide Lizenzen den ausdrücklichen Hinweis, dass sie uneingeschränkt miteinander kombinierbar sind.

Zusammengefasst ist die GNU AGPL also eine zur GPL komplementäre Ergänzung, die das Copyleft auch auf Software anwendet, die nicht mehr in lokalen Installationen, sondern ausschließlich in Form von über Netzwerke übertragenen Services genutzt wird.

Kompatibilität von Copyleft-Lizenzen

Die Entwicklung freier Software lebt davon, auf der Arbeit anderer aufzubauen, also Source Code anderer zu integrieren, zu modifizieren und wieder zu teilen. Stehen alle Teile einer veränderten oder neu zusammengestellten Software unter derselben Copyleft-Lizenz, zum Beispiel der GPLv3, so ist dies auch problemlos möglich: Die Lizenz verlangt ja, dass das Ergebnis unter derselben Lizenz weitergegeben wird.

Kompliziert wird es dann, wenn Software aus Teilen besteht, die unter verschiedenen Lizenzen stehen — hier sind mehrere Faktoren zu beachten.

Kombinierte und abgeleitete Werke

Freie Software entsteht unter zum Teil sehr unterschiedlichen Voraussetzungen und reicht von einer einfachen Fehlerkorrektur bis zu komplexen Projekten mit Millionen Codezeilen. Unabhängig vom Umfang unterscheidet man bei Fragen der Lizenzierung grundsätzlich zwei Arten von Werken: *abgeleitete* (*derivative*) und *kombinierte* (*combined*).

Nehmen wir beispielsweise an, einem Softwareprojekt A fehlt noch eine bestimmte Funktionalität. Statt diese nun von Grund auf selbst zu entwickeln, bietet es sich an, ein anderes Projekt B, das genau diese Funktionalität bietet, mit A zu verbinden. Die Software von B müsste dafür nicht einmal verändert werden, sondern könnte einfach zu A ergänzt werden. Es handelt sich also um ein kombiniertes Werk. Falls sowohl A als auch B unter derselben Copyleft-Lizenz stehen, ergeben sich für das zusammengesetzte Werk keine Probleme.

Stehen A und B unter verschiedenen Copyleft-Lizenzen, so ist bereits Vorsicht geboten: Ist die Kombination von A und B bereits ein eigenes Werk? Und, wenn ja, unter welcher Lizenz kann bzw. muss es dann stehen? Oder ist ein Konflikt dadurch zu umgehen, dass beide Teile A und B mit ihrer jeweiligen Lizenz voneinander getrennt bleiben und kein neues Werk konstituieren?

Noch schwieriger wird es bei abgeleiteten Werken, wenn also Projekt A die Funktionalität von B nur dadurch für sich nutzen kann, dass es Code von B unmittelbar in den Code von A übernimmt. Aus dieser Integration entsteht ein neues, abgeleitetes Werk, dessen Teile nicht mehr voneinander zu trennen sind.

Bei einem abgeleiteten Werk kommt das Copyleft ins Spiel. Steht beispielsweise A unter einer Copyleft-Lizenz wie der GPLv3, so muss auch das neue, abgeleitete Werk gemäß dem Grundsatz der Reziprozität unter der GPLv3 stehen. Was aber, wenn B unter einer anderen Lizenz steht? Ist es eine Copyleft-Lizenz und ist diese kompatibel mit der GPLv3? Oder, wenn es sich gar um einen anderen Lizenztyp handelt, darf B auch unter einer anderen Lizenz veröffentlicht, also relizenziert werden? Oder schließen die Lizenzen von A und B sogar kategorisch einander aus?

Es geht hier nicht darum, die Kombinationsmöglichkeiten und eventuelle Lösungen aufzuzählen. Wichtig ist, die Komplexität des Problems darzustellen, das aus der Verbindung sehr unterschiedlicher Fragestellungen resultiert:

Technisch ist zunächst zu klären, wie die verschiedenen Teile der Software (in unserem Beispiel A und B) zusammenarbeiten: Sind sie voneinander zu trennen oder gibt es bei der Ausführung Prozesse, die man eben nicht mehr eindeutig dem einen oder dem anderen Teil zuordnen kann?

Juristisch stellt sich die Frage, in welchem Verhältnis die Lizenzen von A und B zueinander stehen. Sind sie zueinander kompatibel oder nur in Teilen oder nur in eine Richtung? Ist Relizenzierung eine Option?

Wie gesagt, wir können die Komplexität dieser Fragen hier nur andeuten, nicht auflösen. Grundsätzlich sind Entscheidungen ohne fundierte juristische Kenntnisse nicht möglich, weshalb Lizenzentscheidungen bei neuen, insbesondere aber auch bei kombinierten und abgeleiteten Werken *frühzeitig*, *sorgfältig* und stets nach eingehender *juristischer Beratung* gefällt werden sollten.

Schwaches Copyleft

Das Copyleft hat sich in den vergangenen Jahrzehnten als überaus robust erwiesen, insbesondere in Form der GPL in den Versionen 2 und 3. Besondere technische Anforderungen veranlassen die FSF aber durchaus dazu, mit Anpassungen ihrer Lizenzen zu reagieren. Mit der GNU Affero General Public License haben wir bereits ein solches Beispiel kennengelernt. Weitere Beispiele schauen wir uns in den folgenden Abschnitten an.

Die GNU Lesser General Public License (LGPL)

Ein in der Softwareentwicklung häufig angewendetes Verfahren ist die Nutzung kleiner Module für Standardaufgaben, beispielsweise für das Öffnen oder das Speichern von Dateien. Diese Module — oder auch Sammlungen solcher Module — bezeichnet man als *Softwarebibliotheken*. Es sind meist keine selbständig lauffähigen Anwendungen, sondern Routinen, die die eigentliche Anwendung bei Bedarf integriert. Den Integrationsprozess bezeichnet man als *Linking*, wobei zwei Verfahren unterschieden werden.

Bei *statischen Bibliotheken* übernimmt die eigentliche Anwendung (über Hilfsprogramme und Zwischenschritte) Code der Module fest in den Code der Anwendung. Demgegenüber bindet bei *dynamischen Bibliotheken* die Andwendung ein Modul erst bei Bedarf zur Laufzeit ein und lädt es in den Arbeitsspeicher.

Damit stellt sich in Bezug auf das Copyleft und die Lizenzierung allgemein die Frage: Welche lizenzrechtlichen Implikationen hat das Linking? Macht es diese Form der Übernahme von Code beispielsweise notwendig, dass eine Anwendung, die eine unter GPL stehende Bibliothek nutzt, selbst automatisch der GPL unterliegt? Und gilt dies sowohl für das statische wie auch das dynamische Linking?

Das Verfahren des Linking ist in der Softwareentwicklung so allgegenwärtig und die durch das reziproke Copyleft verbundenen Fragen so virulent, dass die FSF mit der *GNU Lesser General Public License* (LGPL) frühzeitig auch eine lizenzrechtliche Lösung sucht. So entstehen — jeweils

zeitgleich mit den Neufassungen der GPL — auch Aktualisierungen der LGPL. Version 1 verweist auch im Namen noch auf das ursprüngliche Problem: *GNU Library General Public License*. Erst in den Versionen 2 und 3 ist "Library" durch "Lesser" ersetzt und deutet an, um was es inhatlich geht, nämlich eine pragmatische Abschwächung des Copyleft-Prinzips.

Eine unter der LGPL lizenzierte Bibliothek kann demnach von einer Software genutzt werden, ohne dass diese Software dann automatisch selbst dem Copyleft unterliegt. Insbesondere Softwareprojekte unter sogenannten *permissiven* Lizenzen profitieren von diesem Kompromiss, schafft er doch Rechtssicherheit für die Verbindung von lizenzrechtlich per se nicht kompatiblen Ansätzen.

Für LGPL-lizenzierte Software gilt allerdings nach wie vor, dass Veränderungen an dieser Software selbst dem Copyleft unterliegen, also ebenfalls unter der LGPL stehen müssen.

Bedingung ist allerdings, dass die LGPL-lizenzierte Bibliothek austauschbar ist, dass der Benutzer der Software also die Möglichkeit hat, die Bibliothek durch eine veränderte Variante zu ersetzen. Dafür müssen entsprechende Voraussetzungen geschaffen werden, also unter anderem Informationen bereitgestellt werden, wie ein solcher Austausch vorgenommen werden kann.

Andere Copyleft-Lizenzen

Auch andere FOSS-Projekte und -Organisationen suchen die für ihre Bedürfnisse besten rechtlichen Rahmenbedingungen und entwickeln eigene Lizenzen. Ein Beispiel ist die 1998 gegründete *Mozilla Foundation*, heute insbesondere bekannt für die beiden von ihr betreuten Projekte: den Internet Browser *Firefox* und den E-Mail Client *Thunderbird*.

Version 1 der *Mozilla Public License* (MPL) erscheint 1998, die heute (Stand 2024) aktuelle Version 2.0 im Jahr 2012.

Wie die LGPL wird die MPL oft auch als Lizenz mit "schwachem Copyleft" bezeichnet. Tatsächlich sucht sie den Ausgleich zwischen den strikten Forderungen des Copyleft und den Integrationsmöglichkeiten auch mit kommerziellen Produkten. Dies erreicht sie unter anderem durch ein als *File-Level Copyleft* bezeichnetes Prinzip: Nimmt man eine Veränderung an einer Datei vor, die zu einer Software unter der MPL gehört, so kann man diese Datei sogar in eine proprietäre Software integrieren, solange die so veränderte Datei selbst wieder unter MPL steht und damit zugänglich ist.

Ein weiteres Beispiel für eine schwache Copyleft-Lizenz ist die *Eclipse Public License* (EPL) der *Eclipse Foundation*. Die aktuelle Version 2.0 aus dem Jahr 2017 ist der MPL sehr ähnlich und wird häufig auch als die "business-freundlichste" Ausgestaltung einer Copyleft-Lizenz bezeichnet. Oft sind die verschiedenen Copyleft-Lizenzen — es gibt noch weitere als die hier genannten — vor allem jedoch aus der historischen Entwicklung und weniger aus deutlichen juristischen

Unterschieden zu erklären.

Geführte Übungen

1. Wofür steht die Abkürzung GPL?

2. Warum bezeichnet man Copyleft-Lizenzen auch als reziprok?

3. Welche Copyleft-Lizenz der FSF bietet sich für Softwarebibiotheken an?

4. Welche englischen Begriffe ersetzen den Begriff “distribution” in der GPLv3 und warum?

5. Welche der folgenden Copyleft-Lizenzen wurden von der Free Software Foundation herausgegeben?

GPL version 3	
AGPL version 1	
LGPL version 2	
MPL 2.0	
EPL version 2	

Offene Übungen

1. Welche der folgenden sind Copyleft-Lizenzen?

GPL version 2	
3-clause BSD License	
LGPL version 3	
CC BY-ND	
EPL version 2	

2. Darf man Teile zweier Softwareprojekte, die unter verschiedenen Lizenzen mit starkem Copyleft stehen, grundsätzlich zu einem abgeleiteten Werk verbinden? Begründen Sie!

3. Welche der folgenden Lizenzen haben ein starkes Copyleft, welche ein schwaches?

Common Development and Distribution License (CDDL) 1.1	
GNU AGPLv3	
Microsoft Reciprocal License (MS-RL)	
IBM Public License (IPL) 1.0	
Sleepycat License	

4. Beschreiben Sie einige Kompatibilitätsprobleme, die bei der Verbindung von Software unter einer Lizenz mit schwachem Copyleft und Software unter einer Nicht-Copyleft-Lizenz entstehen können!

Zusammenfassung

Diese Lektion behandelt die Eigenschaften von Softwarelizenzen, die dem Grundsatz des Copyleft folgen. In den 1980er Jahren von der Free Software Foundation entwickelt, ist die GNU General Public License (GPL) die derzeit beliebteste Lizenz mit starkem Copyleft. Trotz einiger Überarbeitungen bis zur aktuellen Version 3 sind ihre Grundforderungen nahezu unverändert geblieben: Die durch die Lizenz gewährten Freiheiten, die Software uneingeschränkt nutzen, weitergeben und verändern zu können, müssen stets gewahrt bleiben. Das bedeutet: Die veränderte Software darf nur unter denselben Bedingungen (derselben Lizenz) weitergegeben werden.

Technische Neuerungen wie auch der Wunsch nach Rechtssicherheit bei der Zusammenarbeit mit anderen Projekten veranlassten die FSF zur Entwicklung ergänzender oder alternativer Lizenzen: So trägt die GNU Lesser General Public License (LGPL) dem in der Softwareentwicklung häufig genutzten Verfahren des statischen oder dynamischen Linkens von Softwarebibliotheken Rechnung. Die GNU Affero General Public License (AGPL) reagiert wiederum auf den technischen Umstand, dass Software häufig nur noch in Form von Services über das Netzwerk (insbesondere das Internet), also nicht in lokalen Installationen genutzt wird.

Neben der FSF haben auch andere Projekte, beispielsweise die Mozilla Foundation oder die Eclipse Foundation, Copyleft-Lizenzen entwickelt. Auch diese suchen durch ein schwächeres Copyleft einen Kompromiss zwischen der Sicherung der durch die Lizenz gewährten Freiheiten und einer einfacheren Verbindung mit Software unter anderen Lizenzen.

Grundsätzlich ist Lizenzkompatibilität ein wichtiges Thema bei Copyleft-Lizenzen, denn das Prinzip freier, kollaborativer Softwareentwicklung ist mit den durch die jeweilige Lizenz bestimmten rechtlichen Bedingungen in Einklang zu bringen. Bei jeder Form der Kombination von Software unter verschiedenen Lizenzen sind die rechtlichen Bedingungen im Einzelfall genau zu prüfen.

Antworten zu den geführten Übungen

1. Wofür steht die Abkürzung GPL?

   General Public License

2. Warum bezeichnet man Copyleft-Lizenzen auch als reziprok?

   Das Prinzip des Copyleft verlangt, dass Freiheiten, die durch eine Lizenz gewährt werden, uneingeschränkt auch anderen gewährt werden müssen. Nimmt man zum Beispiel eine Änderung an einer unter der GPL stehenden Software vor, so ist man im Sinne dieser Wechselseitigkeit dazu verpflichtet, diese Änderungen unter denselben Bedingungen anderen zugänglich zu machen.

3. Welche Copyleft-Lizenz der FSF bietet sich für Softwarebibiotheken an?

   GNU Lesser General Public License (GNU LGPL)

4. Welche englischen Begriffe ersetzen den Begriff “distribution” in der GPLv3 und warum?

   Es sind die Begriffe “convey” und “propagate”. Hintergrund ist, dass der Begriff “distribution” fest im internationalen Urheberrecht verankert ist. Um Konflikte oder Missverständnisse zu vermeiden, verzichtet die GPL in Version 3 auf diesen Begriff.

5. Welche der folgenden Copyleft-Lizenzen wurden von der Free Software Foundation herausgegeben?

GPL version 3	X
AGPL version 1	
LGPL version 2	X
MPL 2.0	
EPL version 2	

Antworten zu den offenen Übungen

1. Welche der folgenden sind Copyleft-Lizenzen?

GPL version 2	X
3-clause BSD License	
LGPL version 3	X
CC BY-ND	
EPL version 2	X

2. Darf man Teile zweier Softwareprojekte, die unter verschiedenen Lizenzen mit starkem Copyleft stehen, grundsätzlich zu einem abgeleiteten Werk verbinden? Begründen Sie!

   Nein. Lizenzen mit starkem Copyleft verlangen in der Regel, dass veränderte Versionen unter derselben Lizenz stehen. Das schließt grundsätzlich auch Relizenzierung aus. Die Verbindung von Lizenzen, die beide diesen Grundsätzen folgen, stellt also einen unauflösbaren Widerspruch dar.

3. Welche der folgenden Lizenzen haben ein starkes Copyleft, welche ein schwaches?

Common Development and Distribution License (CDDL) 1.1	schwach
GNU AGPLv3	stark
Microsoft Reciprocal License (MS-RL)	schwach
IBM Public License (IPL) 1.0	schwach
Sleepycat License	stark

4. Beschreiben Sie einige Kompatibilitätsprobleme, die bei der Verbindung von Software unter einer Lizenz mit schwachem Copyleft und Software unter einer Nicht-Copyleft-Lizenz entstehen können!

   Während starkes Copyleft verlangt, dass die Weitergabe der veränderten Software unter derselben Lizenz erfolgt, sind bei schwachem Copyleft die Bedingungen “gelockert”. Dennoch wirft auch hier jede Kombination mit anderen Lizenzen grundsätzliche Fragen der Kompatibilität auf. Bei deren Beantwortung sind wichtige Aspekte zu berücksichtigen: Sind die ursprünglichen Teile der beiden Softwareprojekte in dem neuen Werk klar voneinander getrennt und gegebenfalls weiterhin unterschiedlich zu lizenzieren? Auf welcher Ebene findet die Verbindung der beiden ursprünglichen Softwareprojekte überhaupt statt: Wird Source

Code direkt übernommen oder wird “nur” dynamisch gegen die andere Software (Bibliothek) gelinkt? Erlaubt eine der beiden Lizenzen grundsätzlich Relizenzierung? Alle Fragen dieser Art können nur nach genauer Analyse der jeweiligen Lizenzen und mit fachjuristischer Expertise beantwortet werden.

052.3 Permissive Softwarelizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 052.3

Gewichtung

3

Hauptwissensgebiete

- Verständnis des Konzepts permissiver Softwarelizenzen
- Verständnis der Rechte, die durch permissive Softwarelizenzen gewährt werden
- Verständnis der Verpflichtungen, die durch permissive Softwarelizenzen entstehen
- Verständnis der Haupteigenschaften gängiger permissiver Softwarelizenzen
- Verständnis der Kompatibilität permissiver Softwarelizenzen mit anderen Lizenzen

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- 2-Clause BSD License
- 3-Clause BSD License
- MIT License
- Apache License, version 2.0

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	052 Open-Source-Softwarelizenzen
Lernziel:	052.3 Permissive Softwarelizenzen
Lektion:	1 von 1

Einführung

Im Gegensatz zu *restriktiven* Lizenzen wie der GNU General Public License (GPL) sind *permissive* (“freizügige”) Lizenzen derzeit die am weitesten verbreiteten Open-Source-Lizenzen. Permissive Softwarelizenzen sind in der Regel einfach und flexibel und bieten Urhebern ein breit`es Spektrum an Freiheiten — vielleicht die größte Freiheit unter den Open-Source-Lizenzen.

Diese Art von Lizenz gewährt Softwareentwicklern weitgehende Freiheit bei der Nutzung, Änderung und Weitergabe der Software, solange sie den ursprünglichen Autor benennen. So kann jemand in der Regel ein abgeleitetes Werk unter einer Closed-Source-Lizenz weitergeben, solange das Werk einen Hinweis auf den Autor des Werks enthält, von dem das neue Werk abgeleitet wurde.

Das Prinzip hinter dieser Logik ist die größtmögliche Verbreitung der Software. Permissive Lizenzen wollen Einzelpersonen und der Allgemeinheit zugute kommen, indem sie die kommerzielle Nutzung der Software erleichtern, während die Rechte des ursprünglichen Autors durch Namensnennung gewahrt bleiben.

Es ist kein Zufall, dass die ersten und wichtigsten permissiven Softwarelizenzen im akademischen

Bereich entstanden, wie beispielsweise die BSD-Lizenzen der Universität Berkeley in Kalifornien oder die MIT/X11-Lizenz des Massachusetts Institute of Technology. Diese sind als akademische Open-Source-Lizenzen bekannt. Ihr Einfluss in der Industrie war so groß, dass sie den Grundstein für ähnliche permissive Softwarelizenzen legten, die außerhalb des akademischen Kontextes entwickelt wurden, wie etwa die Apache-Lizenz der Apache Software Foundation.

Rechte und Pflichten permissiver Softwarelizenzen

Im Allgemeinen gewähren die gängigsten permissiven Softwarelizenzen dem Lizenznehmer folgende uneingeschränkte Rechte:

Nutzung

Software unter einer permissiven Softwarelizenz kann von jedermann (von Privatpersonen über kommerzielle Unternehmen bis hin zu Behörden) und für jeden Zweck, ob privat oder beruflich, genutzt werden.

Modifizierung

Die Software kann verbessert, angepasst oder sogar als Komponente (beispielsweise als Bibliothek) in andere Software integriert werden.

Weitergabe

Wenn die Software in Form eines abgeleiteten Werks verändert wird, kann sie unter verschiedenen Lizenzen weitergegeben werden, einschließlich proprietärer Lizenzen.

Die einzige Verpflichtung in permissiven Softwarelizenzen ist in der Regel die der Namensnennung: Der Lizenznehmer ist verpflichtet, den Namen des ursprünglichen Autors der Software im abgeleiteten Werk anzugeben und eine Kopie des Lizenztextes bei jeder Weitergabe der Software aufzunehmen.

Merkmale der wichtigsten permissiven Softwarelizenzen

Dieser Abschnitt behandelt die Unterschiede zwischen der MIT/X11-Lizenz, den gebräuchlichsten BSD-Lizenzen und der Apache-2.0-Lizenz, die heute alle weit verbreitet sind.

MIT/X11-Lizenz

Die MIT/X11-Lizenz (https://opensource.org/license/mit), auch bekannt als X11-Lizenz, ist eine der bereits erwähnten akademischen Lizenzen. Die Lizenz hat ihren Namen von der Software X Window System, die 1987 am Massachusetts Institute of Technology entwickelt wurde. Diese Lizenz ist eine der ältesten und populärsten Softwarelizenzen, auch wegen ihrer einfachen und

klaren Sprache.

Diese Lizenz gewährt dem Lizenznehmer das Recht,

- die Software zu nutzen, zu ändern und zu verbreiten,
- die Software mit oder ohne Änderungen zu vertreiben,
- abgeleitete Werke unter einer anderen Lizenz zu veröffentlichen, auch als Closed Source Software.

Die einzige Verpflichtung des Lizenznehmers besteht darin, den Urheberrechtsvermerk und den Lizenztext in den Quellcode der Software oder der von ihr abgeleiteten Werke aufzunehmen.

Die MIT/X11-Lizenz gilt als kompatibel mit allen wichtigen Copyleft-Lizenzen, sowohl mit schwachen als auch mit starken, insbesondere ist die MIT/X11-Lizenz kompatibel mit der

- GNU General Public License (GPL), Versionen 2.0 und 3.0
- GNU Lesser General Public License (LGPL), Versionen 2.0 und 3.0
- Mozilla Public License (MPL)

Das bedeutet, dass abgeleitete Werke von Software, die ursprünglich unter der MIT/X11-Lizenz lizenziert wurde, unter einer der oben genannten Lizenzen weiterverteilt oder in Projekte aufgenommen werden können, die unter einer der oben genannten Lizenzen veröffentlicht wurden.

2-Clause BSD-Lizenz

Die 2-Clause BSD-Lizenz (https://opensource.org/license/bsd-2-clause) leitet sich von der ursprünglichen BSD-Lizenz ab, die 1980 an der Universität Berkeley in Kalifornien als Lizenz für deren Unix-ähnliches Betriebssystem BSD entstanden ist.

Die Lizenz ist bekannt für ihre Einfachheit und besteht, wie der Name schon sagt, aus nur zwei kurzen Klauseln. Sie ist im Wesentlichen identisch mit der MIT/X11-Lizenz. Wie die MIT/X11-Lizenz verlangt auch sie die Namensnennung in der Software. Darüber hinaus verlangt die 2-Clause BSD-Lizenz, dass der Lizenznehmer den Lizenztext in die Dokumentation und andere Materialien aufnimmt, wenn er den Code weitergibt.

Zusammengefasst gewährt die 2-Clause BSD-Lizenz das Recht,

- die Software zu nutzen, zu ändern und zu verbreiten,
- die Software mit oder ohne Änderungen zu vertreiben,

- abgeleitete Werke unter einer anderen Lizenz zu veröffentlichen, auch als Closed Source Software.

Der Lizenznehmer ist dazu verpflichtet,

- den Urheberrechtshinweis und den Lizenztext in den Quellcode und die Binärdatei der Software oder ihrer abgeleiteten Werke aufzunehmen,
- den Urheberrechtshinweis und den Lizenztext in die Dokumentation oder andere mit der Software gelieferte Materialien aufzunehmen.

Die 2-Clause BSD-Lizenz gilt als kompatibel mit allen wichtigen Copyleft-Lizenzen, sowohl mit schwachen als auch mit starken, insbesondere mit der

- GNU General Public License (GPL), Versionen 2.0 und 3.0
- GNU Lesser General Public License (LGPL), Versionen 2.0 und 3.0
- Mozilla Public License (MPL)

Daher können abgeleitete Werke von Software, die ursprünglich unter der 2-Clause BSD-Lizenz lizenziert wurde, unter einer der oben genannten Lizenzen weiterverbreitet oder in Projekte aufgenommen werden, die unter einer der oben genannten Lizenzen veröffentlicht wurden.

3-Clause BSD-Lizenz

Die 3-Clause BSD-Lizenz (https://opensource.org/license/bsd-3-clause) ist eine weitere Variante der ursprünglichen BSD-Lizenz, die aus nur drei Klauseln besteht. Das Hauptmerkmal, das diese Lizenz von der Schwesterlizenz, der 2-Clause BSD-Lizenz, unterscheidet, ist die “Nichtbefürwortungsklausel” (non-endorsement clause), die verhindert, dass der Name des ursprünglichen Autors für die Verbreitung oder den Verkauf abgeleiteter Werke genutzt wird.

Die 3-Clause BSD-Lizenz gewährt dem Lizenznehmer das Recht,

- die Software zu nutzen, zu ändern und zu verbreiten,
- die Software mit oder ohne Änderungen zu vertreiben,
- abgeleitete Werke unter einer anderen Lizenz zu veröffentlichen, auch als Closed Source Software.

Der Lizenznehmer ist dazu verpflichtet,

- den Urheberrechtsvermerk und den Lizenztext in den Quellcode und die Binärdatei der Software oder ihrer abgeleiteten Werke aufzunehmen,

- den Urheberrechtsvermerk und den Lizenztext in die Dokumentation oder andere mit der Software gelieferte Materialien aufzunehmen,
- den Namen des Urheberrechtsinhabers oder der Mitwirkenden nicht zu nennen, um von dieser Software abgeleitete Produkte zu unterstützen oder zu bewerben, es sei denn, es liegt eine ausdrückliche schriftliche Genehmigung vor.

Um die letztgenannte Verpflichtung, die “Nichtbefürwortungsklausel”, zu erläutern, stellen wir uns ein Szenario vor, in dem ein Unternehmen oder eine Einzelperson ein abgeleitetes Werk erstellt, das auf einer Software basiert, die unter einer permissiven Softwarelizenz steht. Ohne diese Klausel könnte der Lizenznehmer für das neue Werk werben, indem er darauf hinweist, dass es sich um ein abgeleitetes Werk eines bekannten Entwicklers handelt, um von dessen gutem Ruf zu profitieren.

Wie ihre Schwesterlizenz gilt die 3-Clause BSD-Lizenz als kompatibel mit allen wichtigen Copyleft-Lizenzen, sowohl mit schwachen als auch mit starken. Insbesondere ist die 3-Clause BSD-Lizenz kompatibel mit der

- GNU General Public License (GPL), Versionen 2.0 und 3.0
- GNU Lesser General Public License (LGPL), Versionen 2.0 und 3.0
- Mozilla Public License (MPL)

Daher können abgeleitete Werke von Software, die ursprünglich unter der 3-Clause BSD-Lizenz lizenziert wurde, unter einer der oben genannten Lizenzen weiterverbreitet oder in Projekte aufgenommen werden, die unter einer der oben genannten Lizenzen veröffentlicht wurden.

Apache-2.0-Lizenz

Die erste Apache-Lizenz wurde von der Apache Software Foundation, einer 1999 gegründeten gemeinnützigen Organisation, entwickelt, um deren Software zu verbreiten und ihre Einbindung in andere Projekte zu erleichtern. Die Aufgabe bestand darin, die gemeinschaftliche Entwicklung von Software zu gewährleisten und die Open-Source-Philosophie mit einer unternehmensfreundlichen Perspektive zu verbinden.

Die Apache-2.0-Lizenz (https://www.apache.org/licenses/LICENSE-2.0), die wahrscheinlich am weitesten verbreitete unter den permissiven Softwarelizenzen, unterscheidet sich von den zuvor besprochenen Lizenzen in einigen wichtigen Aspekten: Der wichtigste betrifft die Zuweisung von Patentrechten an jeden Lizenznehmer, einschließlich einer Klausel über die Beendigung der Patentlizenz, um Schadensersatzansprüche wegen Patentverletzung zu begrenzen und zu verhindern.

Zwei weitere Verpflichtungen sind von Bedeutung: Der Lizenznehmer muss alle Teile oder Komponenten der Software, die er nicht verändert hat, unter derselben Apache-2.0-Lizenz veröffentlichen. Außerdem muss der Lizenznehmer in allen veränderten Dateien deutliche Hinweise darauf anbringen, dass der Lizenznehmer diese Dateien verändert hat.

Die Free Software Foundation, die sich für restriktive Lizenzen einsetzt, empfiehlt Apache 2.0 als die beste der permissiven Softwarelizenzen für die Verbreitung kleiner Mengen von Software und Bibliotheken.

Die Apache-2.0-Lizenz gewährt dem Lizenznehmer

- das Recht, die Software zu nutzen, zu ändern und zu vertreiben,
- das Recht, die Software mit oder ohne Änderungen zu vermarkten,
- das Recht, abgeleitete Werke unter einer anderen Lizenz zu veröffentlichen, auch als Closed Source Software,
- eine Patentlizenz zur Nutzung, zum Verkauf, zur Einfuhr und zur anderweitigen Weitergabe der Software, die durch ein Patent geschützt ist,
- eine Patentlizenz zur Nutzung, zum Verkauf, zur Einfuhr und zur anderweitigen Weitergabe der Software, was andernfalls den Patentanspruch eines Mitwirkenden verletzen könnte.

Der Lizenznehmer ist dazu verpflichtet,

- in allen veränderten Dateien deutlich darauf hinzuweisen, dass die Dateien verändert wurden,
- alle unveränderten Teile der Originalsoftware unter der Apache-2.0-Lizenz zu veröffentlichen,
- den Urheberrechtsvermerk und den Lizenztext in den Quellcode der Software und ihrer abgeleiteten Werke aufzunehmen,
- den Urheberrechtsvermerk und den Lizenztext in die Dokumentation und andere mit der Software bereitgestellte Materialien aufzunehmen.

Die Apache-2.0-Lizenz gilt nur mit einigen der wichtigsten Copyleft-Lizenzen als kompatibel, insbesondere ist die Apache-2.0-Lizenz kompatibel mit der

- GNU General Public License (GPL), Version 3.0, aber nicht 2.0
- GNU Lesser General Public License (LGPL), Version 3.0, aber nicht 2.0
- Mozilla Public License (MPL), Version 2.0, aber nicht 1.1

Das bedeutet, dass abgeleitete Werke von Software, die ursprünglich unter der Apache-2.0-Lizenz lizenziert wurde, unter einer der oben genannten kompatiblen Lizenzen weiterverbreitet oder in Projekte aufgenommen werden können, die unter einer der oben genannten kompatiblen

Lizenzen veröffentlicht wurden.

Die begrenzte Kompatibilität zwischen der Apache-2.0-Lizenz und anderen Open-Source-Lizenzen ist in erster Linie auf Klauseln zurückzuführen, die sich auf die Gewährung einer Patentlizenz an den Lizenznehmer beziehen.

Permissive Softwarelizenzen im Verhältnis zu anderen Open-Source-Lizenzen

Nachdem wir uns einen Überblick über die gemeinsamen und wesentlichen Merkmale permissiver Softwarelizenzen verschafft haben, untersuchen wir nun, wie sie sich von gemeinfreier Software und von Copyleft-Lizenzen unterscheiden.

Vergleich mit Public Domain Software

Der erste Unterschied zwischen permissiven Softwarelizenzen und der Veröffentlichung von Public Domain (gemeinfreier) Software liegt in ihrer bloßen Existenz. Public Domain ist keine eigentliche Lizenz, sondern nur eine Möglichkeit, Software freizugeben. Mit anderen Worten: Per Definition haben Werke in der Public Domain keine Lizenz.

Der nächste Unterschied liegt in den Verpflichtungen, die permissive Lizenzen dem Lizenznehmer auferlegen. Tatsächlich hat der Benutzer einer Public-Domain-Veröffentlichung keinerlei Verpflichtungen. Wer jedoch Software unter einer permissiven Softwarelizenz nutzt, modifiziert oder weiterverteilt, muss die bereits erläuterte Verpflichtung zur Namensnennung einhalten: Er muss den Namen des ursprünglichen Autors und eine Kopie des Lizenztextes in die Software aufnehmen.

Die Verpflichtung zur Namensnennung hat zur Folge, dass keine abgeleiteten Werke, die von Software unter einer permissiven Softwarelizenz stammen, als gemeinfrei veröffentlicht werden können, da dies das Recht des ursprünglichen Autors auf Namensnennung verletzen (oder zumindest umgehen) würde.

Vergleich mit Copyleft

Die Unterscheidung zwischen permissiven Softwarelizenzen und restriktiven Copyleft-Lizenzen ist komplexer, insbesondere wenn man die Unterschiede zwischen den verschiedenen Copyleft-Lizenzen betrachtet. Wie in den vorangegangenen Lektionen gesehen, gibt es einen grundlegenden Unterschied zwischen *starken* Copyleft-Lizenzen (einschließlich der GNU General Public License (GPL), Versionen 2.0 und 3.0) und *schwachen* Copyleft-Lizenzen (einschließlich der GNU Lesser General Public License (LGPL), Versionen 2.0 und 3.0, und der Mozilla Public License).

Der Hauptunterschied zwischen restriktiven und permissiven Lizenzen liegt im Copyleft-Prinzip, das den Kern restriktiver Lizenzen bildet. Gemäß diesem Prinzip muss der Lizenznehmer, der Software unter einer Copyleft-Lizenz modifiziert und dann weitergibt, das abgeleitete Werk zwangsläufig — ganz oder teilweise — unter derselben Lizenz freigeben wie die ursprüngliche Software. Die Nichteinhaltung dieser Verpflichtung führt zu einer Urheberrechtsverletzung mit den entsprechenden rechtlichen Konsequenzen.

Im Gegensatz dazu erlegen permissive Softwarelizenzen eine solche Verpflichtung nicht auf. Diejenigen, die Software unter dieser Art von Lizenzen verwenden oder freigeben wollen, können die Lizenz für ihre abgeleiteten Werke frei wählen, einschließlich proprietärer Lizenzen.

Starkes Copyleft

Der Unterschied zwischen permissiven Softwarelizenzen und starken Copyleft-Lizenzen ist deutlicher als bei schwachen Copyleft-Lizenzen.

Der wesentliche Unterschied besteht darin, dass starke Copyleft-Lizenzen verlangen, dass abgeleitete Werke unter der gleichen Lizenz wie die ursprüngliche Software veröffentlicht werden. Dies gilt auch, wenn die Copyleft-lizenzierte Software in ein anderes Projekt integriert wird: Das gesamte abgeleitete Werk muss unter der gleichen starken Copyleft-Lizenz veröffentlicht werden.

Schwaches Copyleft

Im Gegensatz zu starken Copyleft-Lizenzen verpflichten schwache Lizenzen nur teilweise, ein abgeleitetes Werk unter derselben Lizenz zu veröffentlichen. Genauer gesagt muss der Lizenznehmer, der unter einer schwachen Copyleft-Lizenz veröffentlichte Software weiterverteilt, dieselbe Lizenz auf den Teil seines Werks anwenden, der von der ursprünglichen Lizenz abgeleitet ist. Beispielsweise muss eine Bibliothek, die auf einer unter der LGPL lizenzierten Bibliothek basiert, ebenfalls unter der LGPL lizenziert werden.

Die schwachen Copyleft-Lizenzen wurden entwickelt, um sie den permissiven Softwarelizenzen anzunähern, unterscheiden sich jedoch von diesen insofern, als letztere unter keinen Umständen eine Verpflichtung zur Beibehaltung der gleichen Lizenz für ein abgeleitetes oder integriertes Werk vorsehen.

Geführte Übungen

1. Welche beiden Verpflichtungen sind in der Regel mit permissiven Softwarelizenzen verbunden?

2. Welche der folgenden Lizenzen ist *keine* permissive Softwarelizenz?

Apache-2.0-Lizenz	
LGPL-Lizenz	
MIT/X11-Lizenz	
3-Clause BSD-Lizenz	

3. Was unterscheidet eine permissive Softwarelizenz von einer Copyleft-Lizenz?

Software, die unter einer Copyleft-Lizenz veröffentlicht wird, darf nicht weitergegeben werden, während Software unter einer permissiven Softwarelizenz weitergegeben werden darf.	
Die von Software unter einer Copyleft-Lizenz abgeleiteten Werke können nicht unter einer proprietären Lizenz veröffentlicht werden, die von Software unter einer permissiven Softwarelizenz abgeleiteten Werke jedoch schon.	
Copyleft-Lizenzen sind nur in den Vereinigten Staaten von Amerika rechtlich anerkannt, während permissive Softwarelizenzen weltweit anerkannt sind.	

4. Welche permissive Softwarelizenz gewährt eine Patentlizenz für die Nutzung, den Verkauf, den Import und die anderweitige Übertragung der durch ein Patent geschützten Software?

Apache-2.0-Lizenz	
2-Clause BSD-Lizenz	
MIT/X11-Lizenz	

3-Clause BSD-Lizenz	

5. Wenn eine Software unter der MIT/X11-Lizenz veröffentlicht wird, können Sie sie dann unter einer proprietären Lizenz weitergeben und ein von der Software abgeleitetes Werk verkaufen?

Offene Übungen

1. Sie modifizieren Software, die unter der Apache-2.0-Lizenz steht, und verbreiten das abgeleitete Werk unter einer proprietären Lizenz weiter. Welche Schritte sollten Sie unternehmen, um die Verpflichtungen der Apache-2.0-Lizenz zu erfüllen?

2. Nennen Sie mindestens drei populäre Projekte, die unter permissiven Softwarelizenzen veröffentlicht wurden.

3. Warum können Sie unter der LGPL-2.0-Lizenz keine Software vertreiben, die Komponenten enthält, die ursprünglich unter der MIT/X11-Lizenz, der Apache-2.0-Lizenz und der 2-Clause BSD-Lizenz veröffentlicht wurden? Unter welcher anderen schwachen Copyleft-Lizenz können Sie die Software veröffentlichen?

Zusammenfassung

In dieser Lektion haben Sie gelernt,

- was permissive Softwarelizenzen sind, welche Rechte sie gewähren und welche Pflichten sie umfassen,
- die Unterschiede zwischen permissiven Softwarelizenzen und anderen Open-Source-Lizenzen,
- Merkmale der gängigsten permissiven Softwarelizenzen,
- Kompatibilität von permissiven Softwarelizenzen mit anderen Open-Source-Lizenzen.

Antworten zu den geführten Übungen

1. Welche beiden Verpflichtungen sind in der Regel mit permissiven Softwarelizenzen verbunden?

   Die Verpflichtung, den Namen des ursprünglichen Autors der Software anzugeben und eine Kopie des Lizenztextes in das abgeleitete Werk aufzunehmen.

2. Welche der folgenden Lizenzen ist *keine* permissive Softwarelizenz?

Apache-2.0-Lizenz	
LGPL-Lizenz	X
MIT/X11-Lizenz	
3-Clause BSD-Lizenz	

3. Was unterscheidet eine permissive Softwarelizenz von einer Copyleft-Lizenz?

Software, die unter einer Copyleft-Lizenz veröffentlicht wird, darf nicht weitergegeben werden, während Software unter einer permissiven Softwarelizenz weitergegeben werden darf.	
Die von Software unter einer Copyleft-Lizenz abgeleiteten Werke können nicht unter einer proprietären Lizenz veröffentlicht werden, die von Software unter einer permissiven Softwarelizenz abgeleiteten Werke jedoch schon.	X
Copyleft-Lizenzen sind nur in den Vereinigten Staaten von Amerika rechtlich anerkannt, während permissive Softwarelizenzen weltweit anerkannt sind.	

4. Welche permissive Softwarelizenz gewährt eine Patentlizenz für die Nutzung, den Verkauf, den Import und die anderweitige Übertragung der durch ein Patent geschützten Software?

Apach- 2.0-Lizenz	X
2-Clause BSD-Lizenz	

MIT/X11-Lizenz	
3-Clause BSD-Lizenz	

5. Wenn eine Software unter der MIT/X11-Lizenz veröffentlicht wird, können Sie sie dann unter einer proprietären Lizenz weitergeben und ein von der Software abgeleitetes Werk verkaufen?

   Ja.

Antworten zu den offenen Übungen

1. Sie modifizieren Software, die unter der Apache-2.0-Lizenz steht, und verbreiten das abgeleitete Werk unter einer proprietären Lizenz weiter. Welche Schritte sollten Sie unternehmen, um die Verpflichtungen der Apache-2.0-Lizenz zu erfüllen?

   Sie sollten

   - in jeder veränderten Datei einen Hinweis einfügen, dass die Datei verändert wurden,
   - alle unveränderten Teile der Originalsoftware unter der Apache-2.0-Lizenz veröffentlichen,
   - den Copyright-Vermerk und den Lizenztext in den Quellcode der Software oder von ihr abgeleiteter Werke einfügen,
   - den Copyright-Vermerk und den Lizenztext in die Dokumentation und andere Materialien aufnehmen, die mit der Software verbreitet werden.

2. Nennen Sie mindestens drei populäre Projekte, die unter permissiven Softwarelizenzen veröffentlicht wurden.

   - Angular Web Framework — MIT/X11-Lizenz
   - Ruby on Rails — MIT/X11-Lizenz
   - Apache HTTP Server — Apache-2.0-Lizenz
   - Kubernetes — Apache-2.0-Lizenz

3. Warum können Sie unter der LGPL-2.0-Lizenz keine Software vertreiben, die Komponenten enthält, die ursprünglich unter der MIT/X11-Lizenz, der Apache-2.0-Lizenz und der 2-Clause BSD-Lizenz veröffentlicht wurden? Unter welcher anderen schwachen Copyleft-Lizenz können Sie die Software veröffentlichen?

   Obwohl die MIT/X11-Lizenz und die 2-Clause BSD-Lizenz mit der LGPL-2.0-Lizenz kompatibel sind, ist die Apache-2.0-Lizenz nicht kompatibel. Software, die Komponenten enthält, die unter der MIT/X11-Lizenz, der Apache-2.0-Lizenz und der 2-Clause BSD-Lizenz veröffentlicht wurden, kann unter der LGPL-3.0-Lizenz veröffentlicht werden, da sie mit all diesen permissiven Softwarelizenzen kompatibel ist.

Linux Professional Institute

Thema 053: Open-Content-Lizenzen

053.1 Konzepte von Open-Content-Lizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 053.1

Gewichtung

2

Hauptwissensgebiete

- Verständnis der Arten von offenen Inhalten
- Verständnis, was Inhalte sind, die dem Urheberrecht unterliegen
- Verständnis für abgeleitete Werke von urheberrechtlich geschütztem Material
- Verständnis für die Notwendigkeit von Lizenzen für offene Inhalte
- Bewusstsein für Marken

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Dokumentation
- Bilder
- Kunstwerke
- Karten
- Musik
- Videos
- Hardware-Designs und Spezifikationen
- Datenbanken
- Datenströme

- Data Feeds

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	053 Open-Content-Lizenzen
Lernziel:	053.1 Konzepte von Open-Content-Lizenzen
Lektion:	1 von 1

Einführung

Stellen Sie sich folgende Situation vor: Frank ist Schriftsteller, der einige seiner Geschichten jedermann im Internet zugänglich machen möchte, allerdings unter bestimmten Bedingungen. Er möchte, dass sein Werk kostenfrei ist. Er möchte, dass jeder die Geschichten nutzen kann, ohne um Erlaubnis zu fragen, aber seine Arbeit soll nicht kommerziell genutzt werden. Schließlich möchte er, dass er in angemessener Form genannt wird, wenn die Geschichten irgendwie genutzt werden. All dies soll ohne komplizierte juristische Schritte geschehen. Wir werden später sehen, warum ein Schriftsteller all diese Absichten haben könnte.

Emma ist Filmemacherin, die Studierenden beibringt, wie man Kurzfilme produziert. Die Studierenden brauchen unter anderem eine Geschichte, um ihre Arbeiten zu erstellen. Emma kennt Franks Geschichten und denkt, dass sie perfekt für ihre Klasse geeignet sind.

Was kann Frank tun, um Emma zu erlauben, seine Geschichten zu verwenden? Diese und viele andere Situationen sind durch ein *Open-Content-Lizenzmodell* zu lösen. Heutzutage können Millionen von urheberrechtlich geschützten Werken von jedermann ohne ausdrückliche Zustimmung des Rechteinhabers oder Zahlung einer Lizenzgebühr weiterverwendet werden, dank ihrer Verbreitung unter Open-Content-Lizenzen. Ressourcen wie Wikipedia, Flickr,

OpenStreetMap, Unsplash und Jamendo sind einige Beispiele für die vielen Plattformen, die dieses Lizenzmodell nutzen.

Nach einer sehr weit gefassten Definition bezeichnet *Open Content* jedes Werk (beispielsweise Filme, Musik, Bilder, Texte, Datenbanken, Datensätze, Dokumentationen, Karten und Hardware-Designs), dessen freie Nutzung und Weiterverbreitung nach dem Urheberrecht erlaubt ist. Dazu gehören auch ursprünglich geschützte Werke, deren Schutzfrist abgelaufen ist und die daher gemeinfrei sind. Eine etwas engere Definition setzt voraus, dass ein Werk von seinem Urheber ausdrücklich unter eine Open-Content-Lizenz gestellt wurde, die die Nutzung und Weiterverbreitung des Werks ohne Zahlung oder Genehmigung erlaubt. Das Open-Content-Modell steht also nicht im Gegensatz zum Urheberrecht, sondern ergänzt es.

Grundlagen des Urheberrechts

Da Lizenzen für offene Inhalte auf dem Urheberrecht beruhen, müssen Sie einige Aspekte des Urheberrechts kennen, um zu verstehen, warum die Lizenzen benötigt werden und was sie erlauben.

Das Urheberrecht ist Teil einer Rechtskategorie, die als *geistiges Eigentum* bekannt ist. Das Urheberrecht schützt Originalwerke, indem es deren Schöpfern das ausschließliche Recht einräumt, bestimmte Nutzungen ihrer Werke durch andere Personen und Unternehmen zu kontrollieren. Im Allgemeinen bedeutet dies, dass niemand ohne die Erlaubnis des Urheberrechtsinhabers das Werk vervielfältigen, verbreiten, öffentlich aufführen, bearbeiten oder etwas anderes damit tun darf, als es anzusehen oder zu lesen.

Zu den Grundlagen, die wir in diesem Abschnitt untersuchen, gehört, was urheberrechtlich geschützt ist und wer die Rechte kontrolliert und die Erlaubnis zur Wiederverwendung eines urheberrechtlich geschützten Werks erteilen kann, wie es Emma in unserem Beispiel tun möchte.

Wie die *World Intellectual Property Organization* (WIPO) ausdrücklich feststellt:

> Das Urheberrecht schützt zwei Arten von Rechten. Wirtschaftliche Rechte erlauben es den Rechteinhabern, aus der Nutzung ihrer Werke durch andere einen finanziellen Gewinn zu ziehen. Die Persönlichkeitsrechte erlauben es den Autoren und Schöpfern, bestimmte Maßnahmen zu ergreifen, um ihre Verbindung zu ihrem Werk zu bewahren und zu schützen. Der Autor oder Schöpfer kann der Inhaber der wirtschaftlichen Rechte sein, oder diese Rechte können an einen oder mehrere Rechteinhaber übertragen werden. In vielen Ländern ist die Übertragung von Urheberpersönlichkeitsrechten nicht zulässig. Das Urheberrecht schützt nur den Ausdruck von Tatsachen oder Ideen. Es erlaubt dem Urheberrechtsinhaber nicht, die Idee exklusiv zu besitzen oder zu kontrollieren. So kann beispielsweise eine Illustration urheberrechtlich geschützt werden, nicht aber die Idee, die

> ihr zugrunde liegt.
>
> — World Intellectual Property Organisation, Understanding Copyright and Related Rights (inoffizielle deutsche Übersetzung)

Wirtschaftliche Rechte, die durch das Urheberrecht geschützt sind, gelten lange Zeit, in der Regel Jahrzehnte nach dem Tod des Urhebers. Persönlichkeitsrechte verfallen nie.

Das Urheberrecht gewährt Rechte an kreativen Werken, etwa an literarischen und künstlerischen Werken, die einen bestimmten Standard an Originalität erfüllen müssen. Das Werk muss die Schöpfung seines Schöpfers sein und darf nicht von einem anderen Werk kopiert worden sein. (Es wird immer wieder darüber diskutiert, wie viel künstliche Intelligenz in ein Werk einfließen kann, um immer noch als originell zu gelten.)

Das Urheberrecht schützt nur den Ausdruck von Tatsachen oder Ideen. Es erlaubt dem Urheberrechtsinhaber nicht, die Idee exklusiv zu besitzen oder zu kontrollieren. So kann beispielsweise eine Illustration urheberrechtlich geschützt werden, nicht aber die Idee, die ihr zugrunde liegt.

Das Urheberrecht entsteht automatisch, wenn ein Werk geschaffen und in einer greifbaren Form fixiert wird, zum Beispiel ein digitales Kunstwerk oder ein Lied. Das bedeutet, dass die Rechte dem Schöpfer zustehen, ohne dass das Werk formell registriert werden muss.

Die Befürworter der Lizenzierung offener Inhalte wollen eine freie Kultur und die Entwicklung eines digitalen Gemeinguts fördern. Sie halten das Urheberrechtssystem sowohl für die Benutzer als auch für die Urheber für zu restriktiv und starr. Durch die Schaffung benutzerfreundlicher Standardlizenzen vereinfachen die Befürworter offener Inhalte die Nutzung und Verbreitung urheberrechtlich geschützter Werke.

Was kann urheberrechtlich geschützt werden?

Urheberrechtsgesetze sind von Land zu Land unterschiedlich. Es gibt jedoch internationale Abkommen zur Vereinheitlichung des Urheberrechts. Urheberrechtlich geschützt werden können literarische und künstlerische Originalwerke, dazu zählen Werke der Kunst, der Musik, der Fotografie, des Films, des Fernsehens, der Literatur und der Programmierung. Die Regeln, nach denen entschieden wird, was urheberrechtlich geschützt werden kann und wie originell ein Werk ist, sind von Region zu Region unterschiedlich.

Manchmal sind diese Kategorien sehr allgemein und gelten für Werke, die sowohl schöpferische als auch rein funktionale Elemente aufweisen: Ein Kurzfilm beispielsweise gilt als künstlerisches Werk, Teile davon können jedoch nicht urheberrechtlich geschützt sein, wenn sie nicht den Anforderungen an Originalität genügen.

Abgeleitete Werke

Bleiben wir bei der Situation vom Anfang der Lektion: Frank hat beschlossen, seine Geschichten unter einer Open-Content-Lizenz zu veröffentlichen, damit sie unter den von ihm gewählten Bedingungen verwendet werden können. Da die Geschichten unter dieser Lizenz stehen, hat Emma sie in ihrer Klasse eingesetzt und beschlossen, einige der Kurzfilme ihrer Studierenden bei einem Filmfestival zu zeigen. In diesem Fall können die von den Studierenden erstellten Inhalte als *abgeleitete (derivative) Werke* betrachtet werden.

Ein abgeleitetes Werk oder eine *Bearbeitung* ist ein Werk, das auf einem bestehenden Werk beruht, wie beispielsweise eine Übersetzung, ein musikalisches Arrangement, eine Dramatisierung, eine Fiktionalisierung, eine Filmversion, eine Vertonung, eine Kunstreproduktion oder eine andere Form, in die das ursprüngliche Werk umgewandelt oder angepasst wird. Das abgeleitete Werk wird zu einem zweiten, separaten Werk, das in seiner Form vom ersten unabhängig ist. Die Umwandlung, Änderung oder Bearbeitung des Werks muss wesentlich sein, um originell und somit urheberrechtlich geschützt zu sein.

Merkmale von Open-Content-Lizenzen

Jede Open-Content-Lizenz bekräftigt das Urheberrecht des Autors und bestätigt, dass jede Person, die das Werk ohne eine Lizenz des Autors nutzt, gegen das Urheberrecht verstößt. Daher funktionieren solche Lizenzen innerhalb des globalen Urheberrechtssystems, anstatt sie versuchen, es zu umgehen.

Open-Content-Lizenzen sorgen auch dafür, dass der Autor des Werks angemessen gewürdigt wird. Wenn Empfänger das Werk an Dritte weitergeben, sollten sie sicherstellen, dass der ursprüngliche Autor genannt und gewürdigt wird. Wenn Empfänger das Werk verändern, sollte das abgeleitete Werk eindeutig den Autor des Originals nennen und angeben, wo das Original zu finden ist.

Im Gegensatz zu den meisten Urheberrechtslizenzen, die restriktive Bedingungen für die Nutzung des Werks vorsehen, gewähren Open-Content-Lizenzen den Benutzern bestimmte Freiheiten, indem sie ihnen Rechte einräumen. Einige dieser Rechte sind praktisch allen Open-Content-Lizenzen gemeinsam, etwa das Recht, das Werk zu vervielfältigen und zu verbreiten. Je nach Lizenz kann der Benutzer auch das Recht haben, das Werk zu verändern, abgeleitete Werke zu erstellen, das Werk aufzuführen, zu zeigen und die abgeleiteten Werke zu verbreiten.

Open-Content-Lizenzen können abgeleitete Werke kontrollieren. Diese Lizenzen umfassen in der Regel das Recht, ein abgeleitetes Werk zu erstellen und es in beliebigen Medien zu verbreiten. Wenn eine Person ein Gemälde unter einer Open-Content-Lizenz lizenziert, kann auch das Recht erteilt werden, ein anderes Bild darauf aufzubauen. Diese Genehmigungen werden unter der

Bedingung erteilt, dass andere das abgeleitete Werk ebenso wie das Originalwerk frei verwenden dürfen.

Daher stellt eine Open-Content-Lizenz normalerweise sicher, dass abgeleitete Werke unter den Bedingungen derselben Open-Content-Lizenz lizenziert werden. Diese Verpflichtung gilt jedoch nicht, wenn das Werk in einer Zusammenstellung enthalten ist. Wenn beispielsweise eine Person ein Album mit Liedern erstellt, von denen eines unter einer Open-Content-Lizenz steht, müssen nicht alle Lieder unter denselben Bedingungen lizenziert werden.

Ein weiterer wichtiger Aspekt von Open-Content-Lizenzen ist die Kontrolle der kommerziellen Nutzung eines Werks. Personen können ihre Werke unter einer Open-Content-Lizenz lizenzieren und dabei die Rechte auf nichtkommerzielle Zwecke beschränken. Alternativ können Personen auch alle Rechte gewähren, einschließlich des Rechts, das Werk kommerziell zu nutzen.

Open-Content-Lizenzen hindern niemanden daran, mit seiner Arbeit Geld zu verdienen. Wenn ein Werk unter einer nichtkommerziellen Lizenz steht, kann ein Verleger oder eine andere kommerzielle Einrichtung das Werk veröffentlichen, wenn er/sie zuvor eine Vereinbarung mit den Urheberrechtsinhabern trifft. Mit anderen Worten: Auch nach der Freigabe eines Werks auf nichtkommerzieller Basis können Urheber das Urheberrecht an eine gewinnorientierte Einrichtung verkaufen, sofern die weitere nichtkommerzielle Nutzung nicht ausgeschlossen wird.

Bedeutung von Open-Content-Lizenzen

Was sind die Vorteile eines Open-Content-Modells, und warum sollten Menschen eine Open-Content-Lizenz verwenden, um ihre kreativen Arbeiten zu verbreiten, anstatt sich auf das traditionelle Urheberrechtsmodell zu verlassen?

Open-Content-Lizenzen ermöglichen eine größere Verbreitung von Werken, als wenn sie standardmäßig eingeschränkt wären. Daher profitieren neue Künstler von diesen Lizenzen, da sie bekannter und von mehr Menschen wahrgenommen werden können. Durch den Verzicht auf bestimmte Kontrollen (und möglicherweise die damit verbundenen Einnahmen) können die Urheber von Inhalten zu mehr Veranstaltungen eingeladen werden, mehr Sponsoring erhalten, mehr Kooperationen eingehen usw.

Open-Content-Lizenzen können im Internetzeitalter praktisch sein, weil Urheber ihre Werke veröffentlichen können, ohne von einer anderen Person oder Institution abhängig zu sein. Ein Fotograf, der seine Fotos ausstellen möchte, könnte sie beispielsweise auf einer persönlichen Website veröffentlichen. Auf diese Weise kann er sich etablieren und einer breiteren Öffentlichkeit bekannt werden, ohne eine Agentur oder Galerie dafür zu beauftragen.

Durch die Wahl der richtigen Lizenz können Rechteinhaber die Verbreitung ihrer Werke

maximieren und die Kontrolle über deren Vermarktung behalten. Wenn jemand ein Werk kommerziell nutzen möchte, kann sich der Urheber des Inhalts das Recht vorbehalten, die Erlaubnis zu erteilen oder zu verweigern. Selbst wenn anderen die kommerzielle Nutzung untersagt ist, können Rechteinhaber ihr Werk weiterhin kommerziell nutzen.

Neben der Möglichkeit, ein Werk wesentlich weiter zu verbreiten, erhöhen Open-Content-Lizenzen die Rechtssicherheit für die Benutzer und senken die Rechtskosten erheblich.

Es gibt auch Fördermodelle, die von der Verwendung einer nichtkommerziellen Lizenz abhängen. Viele Künstler und Kreative nutzen beispielsweise Crowdfunding, um ihr Werk zu finanzieren, bevor sie es unter einer freien Lizenz veröffentlichen. Andere nutzen ein Modell, bei dem die grundlegenden Inhalte kostenlos sind, aber Extras wie gedruckte Versionen oder ein spezieller Zugang zu einer Website nur zahlenden Kunden zur Verfügung steht.

Open-Content-Lizenzen erlauben es Urhebern, ihre Werke ohne Einschränkungen an jedermann und auf jedem Medium und in jedem Format — wie beispielsweise Websites, Fotokopien, CDs oder Bücher — zu verbreiten.

Open-Content-Lizenzen stellen automatisch eine Lizenz zwischen Urhebern und Benutzern her. Ohne eine Open-Content-Lizenz würde die Weitergabe von Werken über eine andere Online-Quelle eine individuelle vertragliche Vereinbarung zwischen Urhebern und Benutzern erfordern.

Marken und Urheberrechte

Marken (manchmal auch *Warenzeichen* oder *Markenzeichen* genannt) sind wie das Urheberrecht eine Form des geistigen Eigentums. Das Recht, das eine Marke schützt, unterscheidet sich jedoch vom Urheberrecht. Markenzeichen schützen Unternehmensmarken, verwandte Wörter wie Markennamen, Logos, Symbole und sogar Klänge und Farben, die verwendet werden, um bestimmte Waren und Dienstleistungen von anderen zu unterscheiden. Jedes besondere Element, das verwendet wird, um Unternehmen und die verkauften Dienstleistungen oder Produkte zu fördern und von anderen zu unterscheiden, kann eine Marke sein.

Der Inhaber einer Marke kann in der Regel andere daran hindern, die unter die Marke fallenden Elemente zu nutzen, wenn für die Öffentlichkeit keine Eindeutigkeit besteht. Das Markenrecht hilft den Herstellern von Waren und Dienstleistungen, ihren Ruf zu schützen, und schützt die Öffentlichkeit, indem es ähnliche Produkte und Dienstleistungen unterscheidbar macht.

Marken können amtlich eingetragen werden oder automatisch nach dem Gewohnheitsrecht geschützt sein. Das bedeutet, dass eine Marke existiert, sobald sie benutzt wird; aber eine Gewohnheitsrechtsmarke bietet nicht den gleichen rechtlichen Schutz wie eine eingetragene Marke. Deshalb lassen viele Unternehmen ihre Marke eintragen.

Urheberrecht und Marken können nebeneinander bestehen. Wikipedia ist beispielsweise eine eingetragene Marke, und ihr Logo ist als Originalwerk oder -schöpfung ebenfalls durch das Urheberrecht geschützt.

Geführte Übungen

1. Können Open-Content-Lizenzen verwendet werden, um das Urheberrecht zu umgehen?

2. Was versteht man unter Open Content?

3. Was ist ein abgeleitetes Werk?

Offene Übungen

1. Was würden Sie tun, wenn Sie ein Werk veröffentlichen wollen und anderen erlauben, es für beliebige Zwecke zu nutzen, solange sie das Werk unter denselben Rechten und Bedingungen weiterverbreiten?

2. Dürfen Sie kommerzielle Bearbeitungen von Werken verbreiten, die unter einer Open-Content-Lizenz veröffentlicht wurden?

Zusammenfassung

In dieser Lektion haben Sie gelernt:

- Grundlagen des Urheberrechts
- was als urheberrechtsfähiger Inhalt betrachtet werden kann
- was ein abgeleitetes Werk oder eine Bearbeitung ist
- gemeinsame Merkmale von Open-Content-Lizenzen
- Arten von Open Content
- Bedeutung und Vorteile des Open-Content-Lizenzmodells
- Urheberrecht und Marken als Arten von geistigem Eigentum <<< == Antworten zu den geführten Übungen

1. Können Open-Content-Lizenzen verwendet werden, um das Urheberrecht zu umgehen?

   Open-Content-Lizenzen können nicht verwendet werden, um das Urheberrecht zu umgehen. Open-Content-Lizenzen sind eine Art von Urheberrechtslizenzen, die unter bestimmten Bedingungen einige Rechte gewähren, während die ausschließlichen Rechte des Urheberrechtsinhabers gewahrt bleiben.

2. Was versteht man unter Open Content?

   Open Content können alle urheberrechtsfähigen Werke sein, die unter einer Open-Content-Lizenz veröffentlicht werden. Dies können Filme, Musik, Bilder, Texte, Datenbanken, Dokumentationen, Karten, Hardware-Designs und andere Schöpfungen sein. Diese Lizenzen gewähren Rechte wie die Nutzung, die Weiterverbreitung, die Erstellung abgeleiteter Werke und die kommerzielle oder nichtkommerzielle Nutzung ohne besondere Genehmigung.

3. Was ist ein abgeleitetes Werk?

   Ein abgeleitetes Werk, auch Adaption genannt, ist ein Werk, das auf einem bestehenden Werk basiert und bei dem das ursprüngliche Werk verändert oder angepasst wird. Übersetzungen, musikalische Bearbeitungen, Dramatisierungen, Filmversionen, Tonaufnahmen und Kunstreproduktionen sind Beispiele für abgeleitete Werke.

Antworten zu den offenen Übungen

1. Was würden Sie tun, wenn Sie ein Werk veröffentlichen wollen und anderen erlauben, es für beliebige Zwecke zu nutzen, solange sie das Werk unter denselben Rechten und Bedingungen weiterverbreiten?

   Sie können das Werk unter einer Lizenz mit Copyleft zur Verfügung stellen. Auf diese Weise müssen alle abgeleiteten Werke des Originals ebenfalls offen sein, sie müssen also unter derselben oder einer anderen kompatiblen Lizenz veröffentlicht werden.

2. Dürfen Sie kommerzielle Bearbeitungen von Werken verbreiten, die unter einer Open-Content-Lizenz veröffentlicht wurden?

   Das hängt von der Lizenz ab, unter der das Originalwerk verfügbar ist. Wenn die Lizenz besagt, dass Sie abgeleitete Werke für jeden Zweck verwenden dürfen, auch für kommerzielle Zwecke, dann dürfen Sie das.

053.2 Creative-Commons-Lizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 053.2

Gewichtung

2

Hauptwissensgebiete

- Verständnis des Konzepts der Creative-Commons-Lizenzen
- Verständnis der Creative-Commons-Lizenztypen und ihrer Kombinationen
- Verständnis der durch Creative-Commons-Lizenzen gewährten Rechte
- Verständnis der Verpflichtungen, die durch Creative-Commons-Lizenzen entstehen

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramm

- Public Domain Dedication (CC0)
- Creative Commons Attribution (CC BY)
- Creative Commons Attribution-ShareAlike (CC BY-SA)
- Creative Commons Attribution-NonCommercial (CC BY-NC)
- Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)
- Creative Commons Attribution-NoDerivatives (CC BY-ND)
- Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND)

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	053 Open-Content-Lizenzen
Lernziel:	053.2 Creative-Commons-Lizenzen
Lektion:	1 von 1

Einführung

Der Erfolg freier Software seit den 1990er Jahren und der gleichzeitige Siegeszug des Internets weckten auch in anderen Bereichen den Wunsch nach ähnlichen Bedingungen zur Förderung der Entwicklung und Verbreitung kreativer — und damit grundsätzlich dem Urheberrecht unterliegender — Werke. Verschiedene Interessen spiegeln sich darin wider.

Künstler und andere kreativ Schaffende wollen selbst bestimmen, unter welchen Bedingungen ihre Werke von anderen genutzt werden können. Gleichzeitig haben sie aber auch ein Interesse daran, die Werke anderer für sich zu nutzen, wie es kreative Prozesse seit jeher tun, etwa durch Zitate, Bearbeitungen oder Adaptionen. Für die Rezipienten der Werke stellen sich Fragen nach den Nutzungsmöglichkeiten, etwa ob und in welcher Form ein Werk vervielfältigt oder weitergegeben werden darf.

Neben der Klärung solcher praktischer Fragen müssen die Regelungen so getroffen werden, dass sie mit den geltenden Rechtsnormen — vor allem dem Urheberrecht — in Einklang stehen, was durch die unterschiedlichen internationalen Regelungen des Urheberrechts erschwert wird.

Nicht zuletzt sollen die Regeln leicht verständlich und einfach anzuwenden sein, um kreative

Prozesse zu beschleunigen und Urhebern und Empfängern Rechtssicherheit zu geben.

Ursprung und Ziele von Creative Commons

Die frühen Lizenzen im Bereich der freien Software, allen voran die GNU General Public License, hatten Pionierarbeit geleistet, da es ihnen gelungen war, einen verlässlichen rechtlichen Rahmen für die völlig neuen Prozesse rund um die gemeinschaftliche Entwicklung von Software zu schaffen.

Im Jahr 2001 gründet eine Gruppe um den Juraprofessor Lawrence Lessig von der Stanford Law School eine gemeinnützige Organisation mit dem Namen *Creative Commons* (CC), die sich an freier Software orientiert. Das Ziel der Organisation wird auf der Website des Projekts wie folgt zusammengefasst:

> Creative Commons ist eine weltweite gemeinnützige Organisation, die die gemeinsame Nutzung und Wiederverwendung von Kreativität und Wissen durch die Bereitstellung kostenloser juristischer Instrumente ermöglicht.
>
> — Creative Commons, Website (FAQ)

Mit dem Begriff “Commons”, der auf eine bereits im Mittelalter dokumentierte Form des gemeinschaftlichen Wirtschaftens verweist, betont die Organisation ein historisch nachweisbares, zutiefst menschliches Bedürfnis nach gemeinschaftlichem, am Gemeinwohl orientiertem Handeln. Ihre selbst gestellte Aufgabe ist es, einen modernen Rechtsrahmen für kreatives Arbeiten zu schaffen. Creative Commons überträgt die Forderungen und Lösungen der Bewegung für freie Software auf andere kreative Bereiche wie Literatur, darstellende Kunst (Malerei, Grafik, Fotografie, Video etc.) und Musik, aber auch Dokumentation und wissenschaftliche Arbeiten — grob gesagt, alle Werke, die dem Urheberrecht unterliegen.

Ähnlich wie die Free Software Foundation setzt auch Creative Commons seine Ziele mit Hilfe von *Lizenzen* um: Zusammenstellungen standardisierter, rechtlich verbindlicher Vorgaben, die die Urheber ihren Werken ausdrücklich zuweisen, in der Regel bevor sie sie veröffentlichen.

Das im US-Urheberrecht verankerte Prinzip “all rights reserved” (“alle Rechte vorbehalten”) wird angesichts der ständig wachsenden technischen Möglichkeiten kreativer Prozesse als zu restriktiv empfunden. Die Urheber selbst sind sich oft nicht im Klaren darüber, wie weit sie auf den Werken anderer aufbauen dürfen, ohne die Grenzen des Urheberrechts zu überschreiten und sich im schlimmsten Fall strafbar zu machen. Creative Commons setzt diesem schwerfälligen Prinzip das “some rights reserved” (“einige Rechte vorbehalten”) entgegen: Die Urheber nennen die Rechte, die sie sich vorbehalten — und verzichten damit auf alle anderen.

Aus der Kombination von nur vier einfachen Grundbedingungen (Modulen) ergeben sich

insgesamt sechs Lizenzen, aus denen die Autoren die für ihre Arbeit passende auswählen und zuordnen können.

Die Module der Creative-Commons-Lizenzen

Die vier genannten Module entsprechen vier einfachen Grundentscheidungen, die ein Urheber im Hinblick auf die Nutzung und Verbreitung eines Werkes trifft. Jedes Modul lässt sich durch ein Kürzel aus zwei Buchstaben und ein Symbol darstellen. Die jeweilige Definition ist sehr kurz und auch für juristische Laien verständlich, was wesentlich zur Popularität der CC-Lizenzen beiträgt. Im Einzelfall können die Klauseln jedoch zu offenen Fragen oder Grauzonen führen.

Namensnennung / Attribution (BY)

> Sie müssen angemessene Urheber- und Rechteangaben machen, einen Link zur Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden. Diese Angaben dürfen in jeder angemessenen Art und Weise gemacht werden, allerdings nicht so, dass der Eindruck entsteht, der Lizenzgeber unterstütze gerade Sie oder Ihre Nutzung besonders.
>
> — Creative Commons, Namensnennung

Das englische Wort “by” (“von”) weist darauf hin, dass die Autoren genannt werden müssen; das Werk muss also eindeutig den Autoren zugeordnet werden, wenn es in irgendeiner Form verwendet, vervielfältigt oder weitergegeben wird. Das BY-Modul ist das einzige, das in *allen* CC-Lizenzen obligatorisch ist. Mit anderen Worten, es gibt kein Werk unter den sechs Standard-CC-Lizenzen, das auf die *Namensnennung* (*Attribution*) verzichten kann.

Diese einfache Anforderung kann bei gemeinschaftlichen Projekten, die über das Internet entwickelt werden, zu praktischen Problemen führen. Bei Werken, die von der Online-Enzyklopädie Wikipedia abgeleitet sind, ist es zum Beispiel sehr fraglich, wie die Namensnennung bei Tausenden von Mitwirkenden in “angemessener Art” zu erfolgen hat.

Nicht kommerziell / NonCommercial (NC)

> Sie dürfen das Material nicht für kommerzielle Zwecke nutzen.
>
> — Creative Commons, Nicht kommerziell

Die Intention des Moduls *nicht kommerziell* (engl. *NonCommercial*) scheint auf den ersten Blick klar: Es soll verhindern, dass ein frei verfügbares Werk von anderen für kommerzielle Zwecke vereinnahmt wird. Dies betrifft oft Werke, die mit öffentlichen Mitteln, beispielsweise an Universitäten, entwickelt wurden. Diese sollen vor Kommerzialisierung geschützt werden, um ihre Qualität und langfristige freie Verfügbarkeit zu sichern.

Tatsächlich führt das NC-Modul in der Praxis häufig zu Unsicherheiten, da im Einzelfall keineswegs klar ist, wann eine Nutzung kommerziell ist: Lehrkräfte, die Materialien unter einer CC-Lizenz mit NC-Modul nutzen, befinden sich beispielsweise bereits in einer rechtlichen Grauzone, wenn sie an einer Privatschule oder einer privaten Hochschule unterrichten, für deren Besuch die Studierenden zahlen müssen.

Viele Kritiker des NC-Moduls weisen auch das Argument zurück, dass freie Materialien durch die kommerzielle Nutzung "unfrei" werden, da die Werke nach wie vor *auch* kostenlos erhältlich sind.

Keine Bearbeitung / NoDerivatives (ND)

> Wenn Sie das Material remixen, verändern oder darauf anderweitig direkt aufbauen, dürfen Sie die bearbeitete Fassung des Materials nicht verbreiten.
>
> — Creative Commons, Keine Bearbeitung

Derivatives (kurz: *Derivs*), also abgeleitete Werke oder Bearbeitungen, wurden bereits in anderen Lektionen erwähnt. Im Kontext von Creative Commons bezieht sich der Begriff auch auf die Schaffung neuer Werke auf der Grundlage bestehender, urheberrechtlich geschützter Werke; das können zum Beispiel Lernmaterialien sein, die ein Lehrer für seinen Unterricht anpasst, oder die Übersetzung eines Romans in eine andere Sprache. *Keine Bearbeitung* schließt die Verbreitung solcher Abänderungen oder Bearbeitungen kategorisch aus: Das Werk darf nur in der vom Urheber veröffentlichten Form weitergegeben werden.

Weitergabe unter gleichen Bedingungen / Share Alike (SA)

> Wenn Sie das Material remixen, verändern oder anderweitig direkt darauf aufbauen, dürfen Sie Ihre Beiträge nur unter derselben Lizenz wie das Original verbreiten.
>
> — Creative Commons, Weitergabe unter gleichen Bedingungen

Das Modul *ShareAlike* gibt an, dass ein Werk nur unter den gleichen Bedingungen weitergegeben werden darf, wie sie in der zugewiesenen Lizenz festgelegt sind. SA gilt als Gegenstück zum Copyleft-Prinzip in den Lizenzen für freie Software, das sicherstellt, dass die von der Lizenz gewährten Freiheiten auch für abgeleitete Werke unverändert gelten.

Das Modul SA ist besonders interessant in Kombination mit anderen Modulen, da die SA-Anforderung auch für andere definierte Bedingungen gilt, wie wir bei der Vorstellung der verschiedenen Lizenzen sehen werden.

Die Kernlizenzen von Creative Commons

Die Kombinationen der vorgestellten Module ergeben insgesamt sechs Lizenzen. Es sind nur sechs, weil nicht alle möglichen Kombinationen der vier Module sinnvoll sind: So schließen sich etwa die Forderung, auch veränderte Werke unter den gleichen Bedingungen weiterzugeben (ShareAlike), und das Verbot von Bearbeitungen (NoDerivs) gegenseitig aus. Folglich gibt es keine Lizenz, die beide Module enthält. Da die Urhebernennung ebenfalls Bestandteil aller Lizenzen ist, ergeben sich die sogenannten *Kernlizenzen*, die wir uns im Folgenden anschauen.

Die Liste dieser Lizenzen kann man sich auch als eine Skala von “viele Freiheiten” bis “wenige Freiheiten” vorstellen, da ein Urheber Schritt für Schritt Fragen zu den Nutzungsmöglichkeiten oder deren Einschränkungen in Bezug auf sein Werk beantwortet und so schließlich die gewünschte Lizenz erhält.

Creative Commons Namensnennung (CC BY)

Die Namensnennung des Autors bzw. Urhebers wurde bereits als obligatorischer Bestandteil aller Lizenzen erwähnt, und so umfasst die erste Lizenz, *CC Namensnennung* (*CC Attribution*), nur diesen Baustein. Ein Werk ist somit für jede Form der Nutzung, Veränderung und Verbreitung freigegeben, solange die Anforderung der Namensnennung erfüllt ist.

Figure 5. CC BY Icon

Wenn beispielsweise ein Autor ein Gedicht unter CC BY veröffentlicht hat, könnte es ein Verleger ohne Rücksprache mit dem Autor und ohne finanzielle Verpflichtungen in eine Gedichtsammlung aufnehmen und über den Buchhandel vertreiben, solange dieses Gedicht eindeutig als Werk des Autors gekennzeichnet ist.

Creative Commons Namensnennung — Weitergabe unter gleichen Bedingungen (CC BY-SA)

CC Namensnennung — Weitergabe unter gleichen Bedingungen (*CC Attribution-ShareAlike*) entspricht CC BY, ergänzt es aber um die Anforderungen von *ShareAlike*, also die Verbreitung von veränderten Versionen des Werks unter den gleichen Bedingungen.

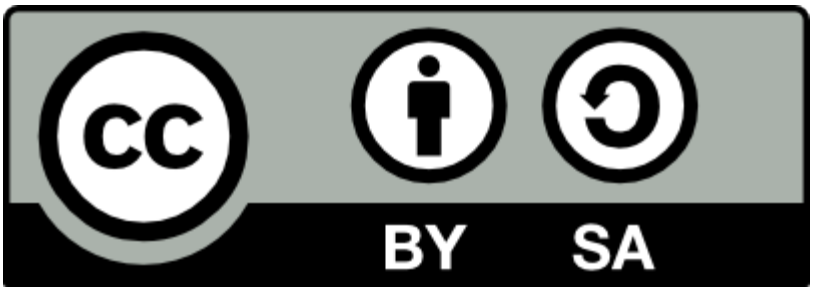

Figure 6. CC BY-SA Icon

Wenn beispielsweise eine Sängerin ihren Song unter die CC BY-SA-Lizenz stellt, kann eine andere Band dieses Werk covern oder adaptieren, sofern sie ihre Version des Songs ihrerseits unter CC BY-SA oder einer anderen, damit kompatiblen Lizenz veröffentlicht.

Creative Commons Namensnennung — Nicht kommerziell (CC BY-NC)

CC Namensnennung — Nicht kommerziell (*CC Attribution-NonCommercial*) ist die erste Lizenz in der Reihe, die die Nutzung eines Werkes mit einem Verbot verknüpft, indem sie die Verbreitung und Bearbeitung zu kommerziellen Zwecken ausschließt.

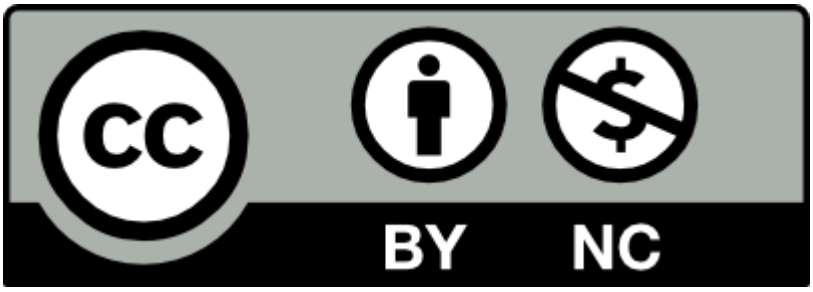

Figure 7. CC BY-NC Icon

In unserem Beispiel wäre es einer Band erlaubt, den Song der Sängerin zu adaptieren; sie dürfte diese Adaption aber nicht kommerziell nutzen, also beispielsweise nicht auf ihren eigenen Platten verkaufen oder auf Konzerten spielen, für die sie eine Gage erhält. Die Band hätte jedoch die Möglichkeit, Adaptionen ihrer Version zusätzlich zu verbieten, also die Adaption unter der Lizenz CC BY-NC-ND (s.u.) zu veröffentlichen.

Creative Commons Namensnennung — Nicht kommerziell — Weitergabe unter gleichen Bedingungen (CC BY-NC-SA)

Im Fall von *CC Namensnennung — Nicht kommerziell — Weitergabe unter gleichen Bedingungen* (*CC Attribution-NonCommercial-ShareAlike*) ist das Verbot der kommerziellen Nutzung an die Verbreitung unter den gleichen Bedingungen geknüpft. Um bei unserem Beispiel zu bleiben: Die Band darf zwar den Song der Sängerin covern, aber sie darf der Coverversion kein ND

hinzufügen, weil das Arrangement unter den gleichen Bedingungen verbreitet werden muss.

Figure 8. CC BY-NC-SA Icon

Creative Commons Namensnennung — Keine Bearbeitung (CC BY-ND)

Wie NonCommercial basiert auch *CC Namensnennung — Keine Bearbeitung* (*CC Attribution-NoDerivs*) auf einem Verbot — in diesem Fall dem Verbot, ein Werk zu verändern oder zu adaptieren. Der Band in unserem Beispiel wäre es daher nicht erlaubt, eine Coverversion des Songs der Sängerin zu verbreiten.

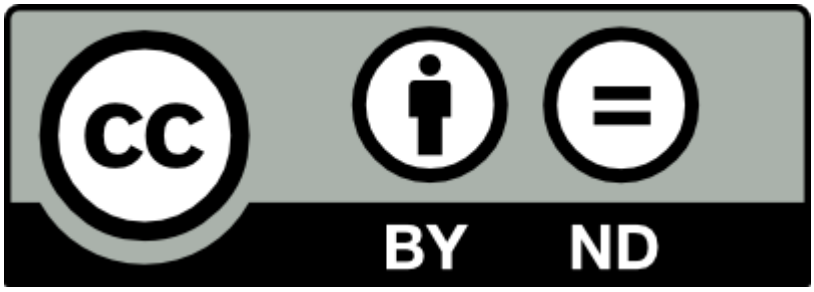

Figure 9. CC BY-ND Icon

Die Verwendung von CC-Lizenzen mit ND-Klausel (also CC BY-ND und die nachfolgend vorgestellte CC BY-NC-ND) wird vor allem im wissenschaftlichen Bereich mit Skepsis betrachtet, da sie den für den wissenschaftlichen Fortschritt notwendigen Wissensaustausch behindern können. Selbst auf der Website von Creative Commons werden Lizenzen mit ND-Modul als unvereinbar mit dem in der Wissenschaft beliebten Prinzip des Open Access bezeichnet. Als Beispiel für die allzu restriktive Tendenz von ND wird angeführt, dass selbst Übersetzungen wissenschaftlicher Artikel als abgeleitete Werke gelten und daher verboten sind.

Creative Commons Namensnennung — Nicht kommerziell — Keine Bearbeitung (CC BY-NC-ND)

Die restriktivste der CC-Lizenzen, *CC Namensnennung — Nicht kommerziell — Keine Bearbeitung* (*CC Attribution-NonCommercial-NoDerivs*), kombiniert das Verbot der kommerziellen Nutzung mit dem Verbot der Verbreitung abgeleiteter Werke. Positiv ausgedrückt bedeutet dies, dass ein Werk

nur in der vom Autor veröffentlichten Form vervielfältigt und verbreitet und nur für nicht kommerzielle Zwecke genutzt werden darf.

Figure 10. CC BY-NC-ND Icon

Die Sängerin in unserem Beispiel hindert also eine Band daran, ihren Song zu covern (ND), riskiert aber gleichzeitig, dass ihr Song von allen Plattformen ausgeschlossen wird, die Einnahmen aus Kunden oder Werbung generieren.

Creative Commons Zero (CC0) und die Public Domain Mark

Es wurde bereits erwähnt, dass das Urheberrecht in verschiedenen Ländern und Rechtsordnungen sehr unterschiedlich geregelt ist. Nach US-amerikanischem Recht kann ein Rechteinhaber beispielsweise vollständig auf sein Urheberrecht verzichten und sein Werk in die so genannte *Public Domain*, also in die allgemeine und uneingeschränkte Nutzung, überführen.

In den meisten europäischen Ländern umfasst das Urheberrecht jedoch neben den reinen Nutzungsrechten auch Persönlichkeitsrechte, die in der Regel nicht übertragbar sind und auf ein Urheber auch nicht verzichten kann. Wenn ein Urheber ein Werk der Öffentlichkeit zugänglich machen möchte, verzichtet er auf alle Nutzungsrechte, bleibt aber Urheber.

Darüber hinaus gibt es Werke, deren Nutzung per se keinen Beschränkungen unterliegt, beispielsweise Werke, deren gesetzliche Schutzfrist abgelaufen ist (zum Beispiel ein Roman 70 Jahre nach dem Tod des Autors), oder Werke, die im Prinzip nie geschützt waren (zum Beispiel Gesetzestexte).

Creative Commons stellt zwei sogenannte *Public Domain Tools* zur Verfügung, um Werke zu kennzeichnen, die der Allgemeinheit ohne Einschränkungen zur Verfügung stehen.

Creative Commons Zero (CC0) knüpft praktisch keine Bedingungen an die Vervielfältigung, Verbreitung und Veränderung eines Werks. Auch die in allen CC-Lizenzen obligatorische Nennung des Urhebers entfällt. Die Lizenz ermöglicht es Rechteinhabern, ihre Werke in möglichst vielen Rechtssystemen als gemeinfrei zu kennzeichnen.

Figure 11. CC0 Icon

Nehmen wir als Beispiel eine Lernunterlage, die für Unterrichtszwecke entwickelt wurde. Der oder die Autoren können das CC0-Label verwenden, um sicherzustellen, dass dieses Material von jedermann uneingeschränkt genutzt werden kann: Es kann ganz oder teilweise kopiert, kostenlos oder gegen eine Gebühr weitergegeben und ganz oder teilweise in andere Dokumente integriert werden. Die ursprünglichen Autoren müssen nirgends genannt werden.

Zur Kennzeichnung von Werken, die keinen urheberrechtlichen Beschränkungen unterliegen, empfiehlt Creative Commons die *Public Domain Mark*.

> Die Public Domain Mark funktioniert wie ein Etikett oder eine Kennzeichnung, die es Institutionen, die über dieses Wissen verfügen, ermöglicht, mitzuteilen, dass ein Werk nicht mehr durch das Urheberrecht eingeschränkt ist und von anderen frei genutzt werden kann.
>
> — Creative Commons, Public Domain Mark

Figure 12. Public Domain Mark Icon

Auswahl von Lizenzen und Kennzeichnung von Werken

Es wurde bereits erwähnt, dass Creative Commons darauf abzielt, sowohl für die Lizenzgeber (Urheber von Inhalten) als auch für die Lizenznehmer (Empfänger der Werke) so einfach wie möglich zu sein. Konkret bietet die Organisation eine Vielzahl von Hilfestellungen von der Auswahl der geeigneten Lizenz bis hin zur Kennzeichnung des Werks und seiner Nutzung.

Der *CC License Chooser* auf der Website der Organisation führt Schritt für Schritt zur Empfehlung der passenden Lizenz, indem er einfache Fragen stellt, die ein Urheber in Bezug auf die

gewünschte Nutzung seines Werks beantwortet (CC License Chooser).

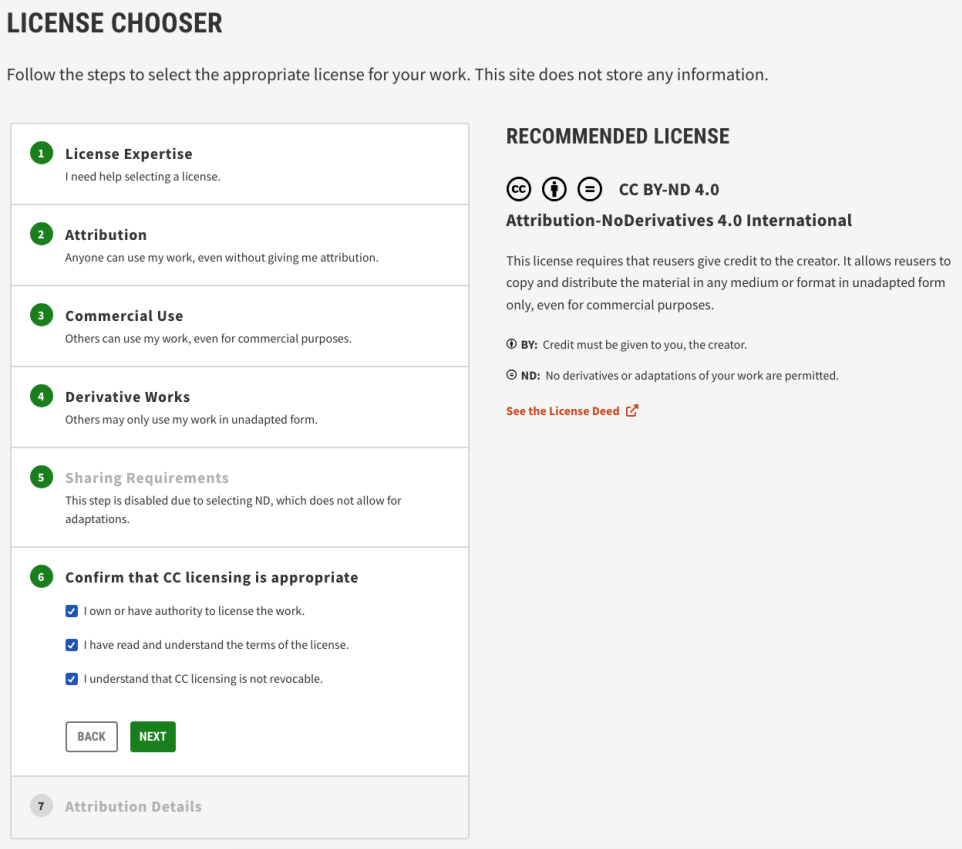

Figure 13. CC License Chooser

Wenn die vorgeschlagene Lizenz den Absichten des Urhebers entspricht, kann er sein Werk entsprechend kennzeichnen. Der License Chooser bietet beispielsweise HTML-Code, den der Urheber direkt in eine Website einfügen kann, auf der das Werk angezeigt wird (HTML-Code mit dem Link zur Lizenz).

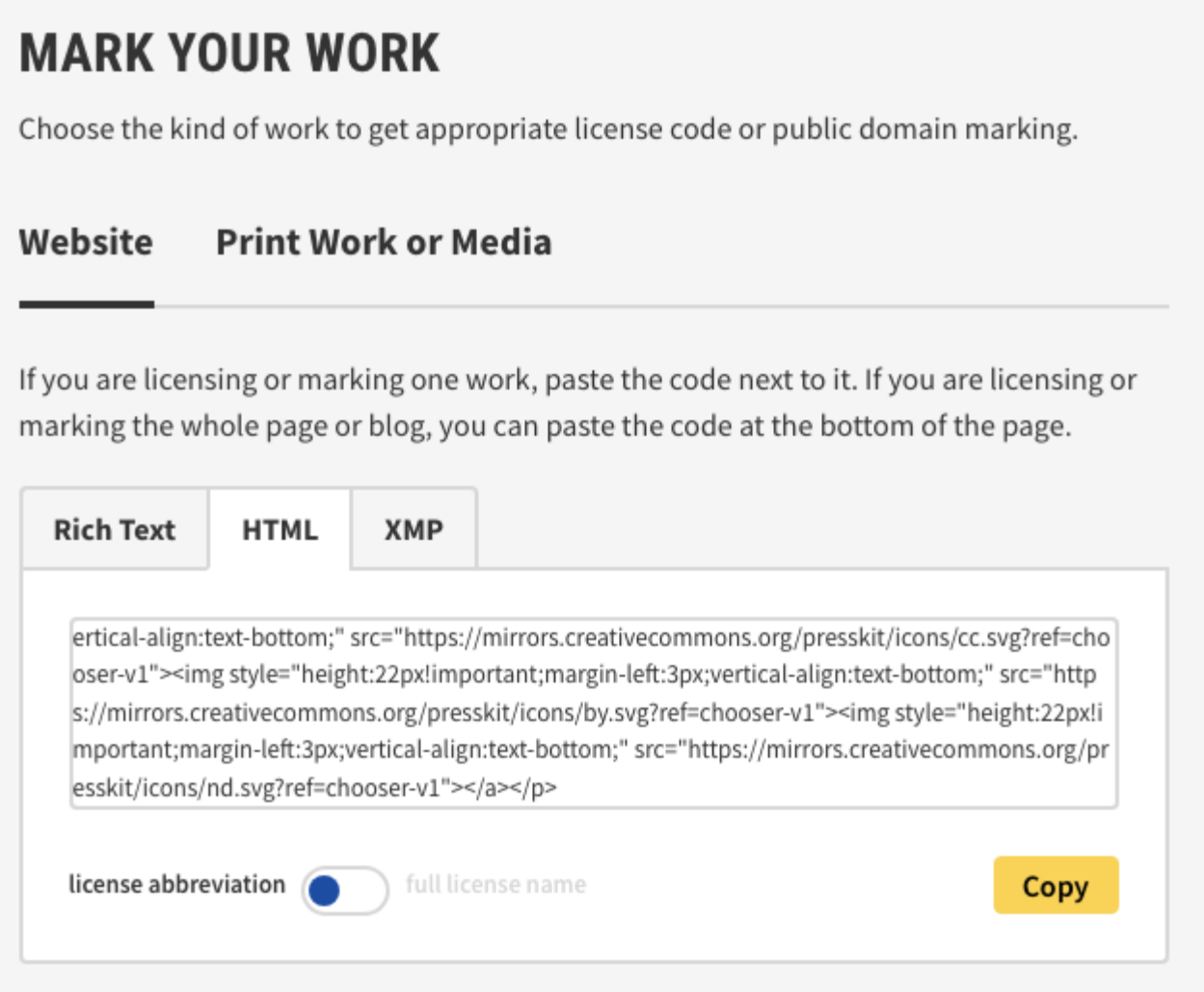

Figure 14. HTML-Code mit dem Link zur Lizenz

Das Ergebnis ist dann ein standardisierter Hinweis mit dem Link zur entsprechenden Lizenz (Lizenzhinweis mit Link zur Lizenz).

Figure 15. Lizenzhinweis mit Link zur Lizenz

Die lizenzspezifischen Icons, in denen die enthaltenen Module direkt sichtbar sind, wurden bereits vorgestellt und haben sich als Blickfang zur Kennzeichnung freier Inhalte im Internet etabliert.

Internationale und portierte Lizenzen

Creative Commons hat sich von Anfang an auf die Entwicklung möglichst allgemeiner, also weltweit gültiger Lizenzen konzentriert, die es mit dem Zusatz “international” (früher auch “unported”) entsprechend kennzeichnet. Andererseits gibt es Varianten, die den Besonderheiten einer regionalen oder nationalen Rechtsordnung Rechnung tragen und als “portiert” (“ported”) bezeichnet werden. So gibt es beispielsweise neben der Lizenz *CC BY-SA 3.0 Unported* auch eine spezielle Version für Deutschland unter dem Namen *CC BY-SA 3.0 Germany*.

Mit der aktuell gültigen Version 4.0 der CC-Lizenzen strebt Creative Commons eine weitere Vereinheitlichung an und hat (bisher) vollständig auf portierte Versionen verzichtet. In der Version 4.0 wird die “internationale” Lizenz ausdrücklich empfohlen:

> Wir empfehlen Ihnen, eine internationale Lizenz der Version 4.0 zu verwenden. Dies ist die aktuellste Version unserer Lizenzen, die nach umfassender Konsultation mit unserem weltweiten Netzwerk von Tochterorganisationen erstellt wurde und international gültig ist. Derzeit gibt es keine Portierungen von 4.0, und es ist geplant, dass nur wenige, wenn überhaupt, erstellt werden.
>
> — Creative Commons, FAQ

Geführte Übungen

1. Welches Modul des Creative-Commons-Lizenzsystems ist Teil aller sechs Creative-Commons-Lizenzen?

2. Welches Modul des Creative-Commons-Lizenzsystems verlangt die Verbreitung eines Werks unter denselben Bedingungen?

3. Wie unterscheidet sich CC0 von den sechs CC-Kernlizenzen?

4. Was ist der Unterschied zwischen “internationalen” und “portierten” CC-Lizenzen?

Offene Übungen

1. Ist es möglich, ein Foto eines gemeinfreien Werks unter eine CC-Lizenz zu stellen?

2. Darf ein Autor seinen Roman, in dem er ein Sonett von Shakespeare vollständig übernimmt, unter einer CC-Lizenz veröffentlichen?

3. Ein Benutzer veröffentlicht ein Foto von sich auf seiner Website unter einer CC-Lizenz. Kann er die Verbreitung dieses Bildes unter Berufung auf seine Persönlichkeitsrechte verhindern?

Zusammenfassung

Die 2001 gegründete Organisation Creative Commons hat sich zum Ziel gesetzt, die gemeinsame Nutzung von kreativen, also urheberrechtlich geschützten Werken mit Hilfe von kostenlosen Rechtsinstrumenten zu unterstützen. Der Schwerpunkt liegt auf vier Modulen (Attribution, NonCommercial, NoDerivs und ShareAlike), die in sechs sogenannten Kernlizenzen zusammengefasst sind und von den Urhebern für ihre Werke ausgewählt werden können.

Antworten zu den geführten Übungen

1. Welches Modul des Creative-Commons-Lizenzsystems ist Teil aller sechs Creative-Commons-Lizenzen?

   Das Modul “Namensnennung” (“Attribution”), abgekürzt “BY”.

2. Welches Modul des Creative-Commons-Lizenzsystems verlangt die Verbreitung eines Werks unter denselben Bedingungen?

   Das Modul “Weitergabe unter gleichen Bedingungen” (“Share Alike”), abgekürzt “SA”.

3. Wie unterscheidet sich CC0 von den sechs CC-Kernlizenzen?

   Mit CC0 sichert sich ein Urheber nicht — wie bei den anderen CC-Lizenzen — bestimmte Rechte, sondern verzichtet auf *alle* Rechte an dem Werk im Rahmen der Möglichkeiten der jeweiligen Rechtsordnung. Dieses Prinzip des “no rights reserved” entspricht der Bezeichnung des Werks als *Public Domain*.

4. Was ist der Unterschied zwischen “internationalen” und “portierten” CC-Lizenzen?

   Bis zur Version 3.0 bot Creative Commons Varianten seiner Lizenzen an, die die Besonderheiten einer regionalen oder nationalen Rechtsprechung berücksichtigten. Seit 4.0 verzichtet CC auf diese sogenannten “ported”-Versionen und empfiehlt die weltweit gültige “international”-Version der jeweiligen Lizenz.

Antworten zu den offenen Übungen

1. Ist es möglich, ein Foto eines gemeinfreien Werks unter eine CC-Lizenz zu stellen?

   Dies hängt davon ab, ob das Foto die Kriterien erfüllt, nach denen es selbst durch das Urheberrecht geschützt ist. Das heißt, das Foto selbst muss als schöpferisches und schützenswertes Werk erkennbar sein, zum Beispiel durch Perspektive, Hintergrund, Filter usw. Ist dies der Fall, kann es unter eine CC-Lizenz gestellt werden, und es gelten dann die Bedingungen der CC-Lizenz. Der ursprüngliche Teil, das eigentliche Motiv, bleibt jedoch gemeinfrei.

2. Darf ein Autor seinen Roman, in dem er ein Sonett von Shakespeare vollständig übernimmt, unter einer CC-Lizenz veröffentlichen?

   Ja, aber nur die eigenen kreativen Teile des Romans unterliegen der vom Autor gewählten CC-Lizenz. Das Sonett, das gemeinfrei ist, bleibt gemeinfrei.

3. Ein Benutzer veröffentlicht ein Foto von sich auf seiner Website unter einer CC-Lizenz. Kann er die Verbreitung dieses Bildes unter Berufung auf seine Persönlichkeitsrechte verhindern?

   Mit der Veröffentlichung eines Fotos unter einer CC-Lizenz erklärt sich der Eigentümer in der Regel mit der Verbreitung einverstanden. Eine Grenze ist dann erreicht, wenn die Nutzung des Fotos die Persönlichkeitsrechte des Eigentümers verletzt, beispielsweise wenn das Bild grob entstellt oder ohne Zustimmung des Eigentümers für Werbung oder in einem politischen Kontext verwendet wird. Die Persönlichkeitsrechte des Eigentümers werden durch die Regelung der Nutzungsrechte durch die CC-Lizenz nicht berührt, so dass er in solchen Fällen gegen die Verwendung seines Fotos juristisch vorgehen kann.

053.3 Andere Open-Content-Lizenzen

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 053.3

Gewichtung

1

Hauptwissensgebiete

- Verständnis der Lizenzierung von Dokumentation
- Verständnis der Lizenzierung von Datensätzen und Datenbanken
- Verständnis der Rechte, die durch Lizenzen für offene Inhalte gewährt werden
- Verständnis der Verpflichtungen, die durch Lizenzen für offene Inhalte entstehen

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- GNU Free Documentation License, Version 1.3 (GFDL)
- Open Data Commons Open Database License (ODbL)
- Community Data License Agreement – Permissive, Version 1.0 (CDLA)
- Community Data License Agreement – Sharing, Version 1.0 (CDLA)
- Open Access

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	053 Open-Content-Lizenzen
Lernziel:	053.3 Andere Open-Content-Lizenzen
Lektion:	1 von 1

Einführung

In den vorangegangenen Lektionen wurden bereits das Konzept von Open Content sowie die Creative-Commons-Lizenzen vorgestellt. Neben Creative-Commons-Lizenzen gibt es jedoch weitere Open-Content-Lizenzen, die teilweise auf bestimmte Inhalte bezogen ausgestaltet sind und regelmäßig auch für “Nebenprodukte” (wie etwa Dokumentation) von Open-Source-Projekten Anwendung finden.

Lizenzen für (Software-)Dokumentation

In großen Software-Repositories sind regelmäßig auch Dokumentation und Manuals zu der Software enthalten. Dokumentation enthält in der Regel verschiedene urheberrechtlich schutzfähige Elemente, wie zum Beispiel erläuternde Texte, veranschaulichende Grafiken oder Code-Beispiele. Open-Source-Lizenzen, die für die Inhalte eines Repositorys gelten, sind grundsätzlich auch auf solche Texte anwendbar, wenn keine anderweitigen Informationen vorliegen; so ist etwa die GPLv3.0 auf “any copyrightable work” anwendbar und nicht auf Code beschränkt.

Oft unterliegt Dokumentation aber davon abweichenden Lizenzbedingungen. Das kann aus

verschiedenen Gründen sinnvoll sein: Meist handelt es sich bei Dokumentation eben nicht um Code, sondern um andere Werkgattungen. Die Lizenzbedingungen einer strengen Copyleft-Lizenz für Computerprogramme verhindert dann möglicherweise die Übernahme einzelner Teile der Dokumentation für andere Programme. Auch ist unklar, was damit gemeint ist, wenn für eine GPLv3.0-lizenzierte Dokumentation der "Quellcode" bereitgestellt werden soll. Zudem war es früher üblich, Dokumentation bzw. Handbücher für Programme auch in gedruckter Fassung zu verbreiten. Diesem Umstand kann durch spezifische Dokumentationslizenzen besser Rechnung getragen werden (wenn zum Beispiel bestimmte Lizenzpflichten an eine bestimmte Anzahl verbreiteter Exmplare geknüpft werden).

Oft kommen bei Dokumentation Creative-Commons-Lizenzen zum Einsatz, aber es gibt auch für Dokumentation spezielle Copyleft-Lizenzen, wie zum Beispiel die *Free Documentation License* (FDL). Diese erklärt in ihrer Präambel zu ihrem Zweck:

> Der Zweck dieser Lizenz ist es, ein Handbuch, ein Lehrbuch oder ein anderes funktionales und nützliches Dokument "frei" im Sinne von Freiheit zu machen: jedem die tatsächliche Freiheit zu sichern, es zu kopieren und weiterzugeben, mit oder ohne Veränderungen, entweder kommerziell oder nicht kommerziell. In zweiter Linie sichert diese Lizenz dem Autor und dem Herausgeber eine Möglichkeit, für ihre Arbeit gewürdigt zu werden, ohne für Änderungen verantwortlich gemacht zu werden, die andere vorgenommenen haben.
>
> Diese Lizenz ist eine Art "Copyleft", was bedeutet, dass abgeleitete Werke des Dokuments selbst im gleichen Sinne frei sein müssen. Sie ergänzt die GNU General Public License, eine Copyleft-Lizenz, die für freie Software entwickelt wurde.
>
> Wir haben diese Lizenz entwickelt, um sie für Handbücher für freie Software zu verwenden, weil freie Software freie Dokumentation braucht: ein freies Programm sollte mit Handbüchern ausgeliefert werden, die die gleichen Freiheiten bieten wie die Software. Aber diese Lizenz ist nicht auf Handbücher beschränkt; sie kann für jedes Textdokument verwendet werden, unabhängig vom Thema oder ob es als gedrucktes Buch veröffentlicht wird. Wir empfehlen diese Lizenz vor allem für Werke, deren Zweck Anleitung oder Referenz ist.
>
> — Free Documentation License, Präambel (inoffizielle deutsche Übersetzung)

Die Lizenz ist sehr spezifisch an Dokumentation angepasst; so adressiert sie etwa auch gedruckte Fassungen. Insbesondere wirkt sich das Copyleft einer Dokumentationslizenz explizit nicht auf den Code aus, mit dem sie ausgeliefert wird. Zwar wäre es auch möglich, eine Dokumentation unter der GPLv3.0 gemeinsam mit MIT-lizenziertem Code zu verbreiten — da in diesem Fall zwei unabhängige Werke vorlägen, würde die GPLv3.0 der Dokumentation nicht dazu führen, dass der Code unter der GPLv3.0 lizenziert werden müsste. Aber eine dedizierte Lizenz für Dokumentation schafft diesbezüglich mehr Klarheit.

Lizenzen für Datenbanken

Auch für Datenbanken gibt es spezielle Lizenzen, die unter anderem den Einsatz einer Datenbank als Ressource für ein Programm sowie die Besonderheiten des urheberrechtlichen Schutzes von Datenbanken berücksichtigen.

Begriff der "Datenbank"

Im Urheberrecht wird in einigen Rechtsordnungen (unter anderem in Europa) unterschieden zwischen Datenbankwerken, die wie andere urheberrechtlich geschützte Werke (Bilder, Texte etc.) Schutz aufgrund einer *persönlichen geistigen Schöpfung* (Auswahl und Anordnung der Elemente) genießen, und Datenbanken, für deren Herstellung (Beschaffung, Überprüfung, Darstellung) eine *wesentliche Investition* erforderlich war (Datenbankherstellerrecht).

Gegenstand des Schutzes ist in beiden Fällen eine Sammlung unabhängiger Elemente (Werke, Daten oder andere unabhängige Elemente), die systematisch oder methodisch angeordnet und einzeln elektronisch oder mit anderen Mitteln zugänglich sind. Auf die urheberrechtliche Schutzfähigkeit der Elemente selbst kommt es dabei nicht an; es kann sich also auch um bloße Zahlenwerte handeln.

Eine Sammlung von Daten kann also als Datenbankwerk oder im Rahmen des Datenbankherstellerrechts (in der EU) urheberrechtlich schutzfähig sein. Für den Schutz eines Datenbankwerkes ist es erforderlich, dass sich in Anordnung und Auswahl der Elemente eine persönliche geistige Schöpfung ausdrückt. Das dürfte für Datensammlungen, die als Grundlage für Software verwendet werden, meist nicht der Fall sein — diese können jedoch im Rahmen des Datenbankherstellerrechts geschützt sein. Dafür muss für die Sammlung und Anordnung der Elemente eine "wesentliche Investition" getätigt worden sein, wobei diese Investition auch in Zeit und Personalressourcen bestehen kann.

Für Datenbanken bzw. Datensammlungen stehen inzwischen verschiedene Lizenzen zur Verfügung, die den Verwendungsfall von Daten besser erfassen als "klassische" Open-Source-Softwarelizenzen. Schauen wir uns die Abgrenzung an einem Beispiel an:

Im Rahmen eines wissenschaftlichen Forschungsprojekts sollen sämtliche Gedichte eines Autors gesammelt und auf einer Website veröffentlicht werden. Dort sind sie alphabetisch nach Titel sortiert und einzeln abrufbar. Allerdings ist für das Projekt erforderlich, dass die entsprechenden Urheberrechte zur Verbreitung und Vervielfältigung der Gedichte bei dem Autor eingeholt werden. Dafür werden Lizenzgebühren bezahlt.

Die Lizenzgebühren können eine "wesentliche Investition" darstellen. Da die Gedichte einzeln abrufbar und systematisch (nach dem Alphabet) angeordnet sind, liegt eine Datenbank vor, die

nach dem Datenbankherstellerrecht schutzfähig ist. Die Elemente (Gedichte) wurden jedoch nicht spezifisch ausgewählt (es ging um Vollständigkeit) und auch die Anordnung ist nicht kreativ, sondern rein systematisch. Eine persönliche geistige Schöpfung in Bezug auf die Datenbank liegt also nicht vor, so dass die Datenbank nicht als Datenbankwerk zu qualifizieren ist.

Wären hingegen Anordnung und Auswahl der Gedichte nach bestimmten Kriterien erfolgt, etwa um bestimmte Motive im Werk des Autors im Kontext der Schaffenszeit zu dokumentieren, könnte eine persönliche geistige Schöpfung vorliegen — und dann wäre diese Sammlung von Gedichten als Datenbankwerk schutzfähig.

Für Lizenzen ergibt sich daraus in der Regel kein wesentlicher Unterschied. Urheberrechtlich werden die beiden Kategorien allerdings unterschiedlich behandelt, insbesondere in der Schutzdauer: für Datenbanken nach dem Datenbankherstellerrecht nur 15 Jahre nach Herstellung/Veröffentlichung im Unterschied zu 70 Jahren nach dem Tod des Autors für Datenbankwerke.

Abgrenzung: Daten

Stets getrennt zu betrachten sind die enthaltenen *Daten* (in unserem Beispiel die Gedichte), die für sich schutzfähig sein können (aber nicht müssen) und demnach auch abweichenden Lizenzbedingungen unterliegen können. So könnte die Sammlung der Gedichte unter der Open Database License (s.u.) stehen, während für die einzelnen Gedichte möglicherweise eine Creative-Commons-Lizenz oder eine proprietäre Lizenz gilt.

Abgrenzung: Database Management System (DBMS)

Urheberrechtlich getrennt zu betrachten ist darüber hinaus die Software zur Verwaltung der Daten, das sogenannte *Database Management System* (DBMS), das im Urheberrecht nicht als Teil der Datenbank gilt, sondern als Mittel, um auf die Daten zuzugreifen. Solche Anwendungen — wie MySQL, PostgreSQL, MariaDB und andere — sind als Computerprogramme gesondert zu betrachten und als solche schutzfähig.

Datensätze zum Training von Machine-Learning-Modellen

Von Webseiten wie huggingface.co oder kaggle.com lassen sich verschiedenste Sammlungen von Daten herunterladen, beispielsweise für das Training von Machine-Learning-Modellen. Solche auch als *Datensatz* bezeichneten Sammlungen unterliegen meist dem Datenbankherstellerrecht, sofern eine “wesentliche Investition” für die Herstellung getätigt wurde (was in der Regel von außen nicht ohne weiteres erkennbar ist).

Der Begriff des “Datensatzes” ist dabei etwas unscharf, weil er zum einen eine Zeile in einer Datenbank bezeichnen kann — also einen “Eintrag” bzw. ein abgrenzbares Element — zum

anderen aber auch Sammlungen von Daten bzw. Datensätzen.

Open Database License (ODbL)

Die genannten Unterscheidungen werden auch in der *Open Database License* (ODbL), einer weit verbreiteten Lizenz im Datenbankbereich, sichtbar. Entwickelt wurde die ODbL von *Open Data Commons*, einem Projekt der Open Knowledge Foundation. Die Lizenz stellt schon in ihrer Präambel klar, dass zu differenzieren ist zwischen dem Schutz der Datenbank und dem Schutz der einzelnen Inhalte der Datenbank. Die Lizenz bezieht sich nur auf ersteres, während die Inhalte davon unabhängig auch anderweitig lizenziert sein können.

Selbstverständlich ist es dennoch möglich, für Datenbank und Inhalte die gleiche Lizenz zu wählen, also etwa eine Sammlung von Werken, die CC-BY-4.0-lizenziert sind, in einer Datenbank zusammenzustellen, die wiederum unter CC-BY-4.0 steht.

Welche Lizenz für die Datenbank gewählt wird, hängt von den Zielen des Datenbankherstellers ab. Um bei unserem Beispiel der Gedichtsammlung zu bleiben: Soll die Gedichtdatenbank beliebig verändert und auch unter anderen Lizenzbedingungen weitergegeben werden dürfen, wäre die ODbL nicht die richtige Wahl: Die ODbL enthält ein Copyleft für Datenbanken, das heißt, dass eine von einer ODbL-lizenzierten Datenbank abgeleitete Datenbank (also etwa eine Datenbank, die alle Elemente und die Anordnung der Elemente der Ausgangsdatenbank übernimmt und weitere hinzufügt) nur zu denselben Bedingungen weiterverbreitet werden darf. In diesem Fall wäre beispielsweise eine Creative-Commons-Lizenz mit Bearbeitungsrechten (beispielsweise CC-BY) zu wählen.

Die ODbL stellt außerdem klar, dass sie sich keinesfalls auf Computerprogramme bezieht, die die Datenbank verwalten (also DBMS): “Diese Lizenz gilt nicht für Computerprogramme, die für die Erstellung oder den Betrieb der Datenbank verwendet werden.”

Sie regelt aber, dass ein Werk, das aus den Inhalten der Datenbank entstanden und öffentlich nutzbar ist (ein sogenanntes *produced work*) zumindest darauf hinweisen muss, welche Datenbank dafür genutzt wurde und dass diese unter den Bedingungen der ODbL zur Verfügung steht. Ein Beispiel für ein solches *produced work* ist etwa eine Straßenkarte, die aus den in der Datenbank gesammelten Koordinaten erstellt wird.

Wenn eine der bekannten (softwarespezifischen) Copyleft-Lizenzen wie etwa die GPLv3.0 für eine Datenbank verwendet würde, ist davon auszugehen, dass auch ein Computerprogramm, das die Datenbank nutzt (etwa für ein Shopsystem) nur zu den Bedingungen der GPLv3.0 verbreitet werden dürfte. Die Lizenz eines DBMS dürfte hingegen unabhängig von der Lizenz der Datenbank sein, weil das DBMS nicht als “abgeleitetes Werk” der Datenbank betrachtet wird; es ermöglicht lediglich den Zugriff und die Bearbeitung und stellt damit eher ein unabhängiges Werk dar.

Community Data License Agreements (CDLA)

Die *Community Data License Agreements* (CDLA), ein Projekt der Linux Foundation, konzentrieren sich ebenfalls auf die Lizenzierung von Daten (in Abgrenzung von Software); auch Trainingsdatensammlungen für Machine-Learning-Systeme können Gegenstand von CDLA sein.

> Unsere Communities wollten Datenlizenzvereinbarungen entwickeln, die eine gemeinsame Nutzung von Daten ermöglichen, ähnlich wie bei Open Source Software. Das Ergebnis ist eine groß angelegte Zusammenarbeit bei Lizenzen für die gemeinsame Nutzung von Daten innerhalb eines rechtlichen Rahmens, den wir Community Data License Agreement (CDLA) nennen.
>
> Diese Lizenzen bilden den Rahmen für die gemeinsame Nutzung von Daten, der sich in Open Source Software Communities bewährt hat.
>
> — CDLA Website (cdla.dev)

Den CDLA liegt also ein Framework zugrunde, das verschiedene Ebenen von Lizenzfragen berücksichtigt, insbesondere wenn Daten aus mehreren Quellen von verschiedenen Beiträgern zusammengestellt werden. In unserem Beispiel könnte das eine Sammlung sein, in der verschiedene Autoren ihre eigenen Gedichte beisteuern, um eine gemeinsame Gedichtdatenbank aufzubauen.

So unterscheidet CDLA zwischen der “inbound license” für Daten, die zur Sammlung hinzugefügt werden, und der “outbound license” für die Verwendung der Daten. Auf einer dritten Ebene macht sie Vorgaben für die Organisation, die die Daten hostet bzw. bereithält.

Um im Beispiel zu bleiben: Ein Autor, der sein Gedicht beiträgt, würde sich an den Vorgaben für “inbound licenses” orientieren. Die “outbound license” betrifft die Bedingungen, auf die sich die Herausgeber der Gedichtsammlung insgesamt verständigt haben.

Ähnlich wie bei den Open-Source-Lizenzen für Software stehen CDLA der Kategorien *permissiv* und *sharing* zur Verfügung, die wir im Folgenden kurz vorstellen. Einen Eindruck, welche Lizenzen für Datenbanken bzw. Datensätze verwendet werden, vermittelt ein Besuch der bereits genannten Webseiten kaggle.com oder huggingface.co: Über Filter lässt sich dort beispielsweise herausfinden, welche Datensammlungen unter CDLA angeboten werden.

Community Data License Agreement — Permissive

Das permissive CDLA steht seit 2021 bereits in Version 2.0 zur Verfügung und ist gegenüber Version 1.0 deutlich knapper und übersichtlicher formuliert.

In Version 1.0 werden Rechte zur Nutzung und Veröffentlichung der lizenzierten Daten

eingeräumt, inklusive der Veröffentlichung sogenannter “Enhanced Data”, was auch Bearbeitungen oder Ergänzungen zum Datenset umfasst. Bei Weitergabe bzw. Veröffentlichung der Daten sind grundlegende Lizenzpflichten einzuhalten; so ist unter anderem der Lizenztext weiterzugeben, Veränderungen sind zu kennzeichnen und Urhebervermerke sind beizubehalten. Die Daten können darüber hinaus aber auch unter zusätzlichen oder anderen Lizenzbedingungen weitergegeben werden.

In Version 2.0 entfällt die Formulierung “Enhanced Data”, stattdessen wird die Rechteeinräumung präziser formuliert:

> Ein Datenempfänger darf die von dem/den Datenanbieter(n) im Rahmen dieser Vereinbarung zur Verfügung gestellten Daten nutzen, verändern und weitergeben, wenn er die Bedingungen dieser Vereinbarung einhält.
>
> — Community Data License Agreement - Permissive, version 2.0 (inoffizielle deutsche Übersetzung)

Einzige Bedingung für das Teilen (“share”) der Daten ist in Version 2.0 die Weitergabe des Lizenztextes bzw. eines Links zum Lizenztext.

Community Data License Agreement — Sharing

Die Sharing-Variante der CDLA, die bislang nur in Version 1.0 vorliegt, enthält ein Copyleft für Daten dergestalt, dass die Weitergabe (“Publish”) von “Enhanced Data” nur zu denselben Bedingungen gestattet ist:

> Die Daten (einschließlich der erweiterten Daten) müssen im Rahmen dieser Vereinbarung gemäß diesem Abschnitt 3 veröffentlicht werden;
>
> — Community Data License Agreement -- Sharing, version 1.0 (inoffizielle deutsche Übersetzung)

Open Access

Ein Begriff, der im Zusammenhang mit Open Content häufig fällt, ist *Open Access*. Auch wenn er im juristischen Sinne nicht definiert ist, so macht die Erklärung der *Budapest Open Access Initiative* (BOAI) die Absicht hinter Open Access klar. Die Initiative entstand aus einer Tagung in Budapest im Jahr 2001 und fasst die internationalen Bemühungen um Open Access zusammen.

> Die Literatur, die online frei zugänglich sein sollte, ist diejenige, die Wissenschaftler der Welt zur Verfügung stellen, ohne dafür ein Entgelt zu erwarten. Dazu gehören in erster Linie ihre begutachteten Zeitschriftenartikel, aber auch alle nicht begutachteten Vorabdrucke, die sie online stellen möchten, um Kommentare abzugeben oder um Kollegen auf wichtige Forschungsergebnisse aufmerksam zu machen. Es gibt viele Stufen und Arten eines

> breiteren und leichteren Zugangs zu dieser Literatur. Mit “Open Access” zu dieser Literatur meinen wir die freie Verfügbarkeit im öffentlichen Internet, die es allen Benutzern erlaubt, die Volltexte dieser Artikel zu lesen, herunterzuladen, zu kopieren, zu verbreiten, zu drucken, zu durchsuchen oder mit ihnen zu verlinken, sie für die Indexierung zu crawlen, sie als Daten an Software weiterzugeben oder sie für jeden anderen rechtmäßigen Zweck zu verwenden, ohne andere finanzielle, rechtliche oder technische Hindernisse als die, die mit dem Zugang zum Internet selbst verbunden sind. Die einzige Einschränkung bei der Vervielfältigung und Verbreitung und die einzige Rolle des Urheberrechts in diesem Bereich sollte darin bestehen, den Autoren die Kontrolle über die Integrität ihrer Arbeit und das Recht auf angemessene Anerkennung und Zitierung zu geben.
>
> — Erklärung der Budapest Open Access Initiative (inoffizielle deutsche Übersetzung)

Mit Open Access ist demnach insbesondere der freie Zugang zu wissenschaftlicher Literatur und anderen Inhalten im Internet gemeint. Die Open-Access-Bewegung hat ihren Ursprung in den 1990er Jahren, mit dem Ziel, insbesondere wissenschaftliche Veröffentlichungen der Allgemeinheit verfügbar zu machen. Open Access bezeichnet also nicht eine spezifische Lizenz oder bestimmte Lizenzbedingungen, sondern vielmehr ein Konzept der Lizenzierung.

Wenn ein Beitrag mit Open Access veröffentlicht wird, erfolgt zunächst die Wahl einer Open-Content-Lizenz wie zum Beispiel einer der Creative-Commons-Lizenzen. Die Wahl einer CC-Lizenz ist für Open Access aber nicht zwingend. Über die Jahre hat sich eine Kategorisierung verschiedener Open-Access-Strategien entwickelt, die auch duale Lizenzierung berücksichtigen — also etwa das Erteilen einer umfassenden Lizenz an einen Verlag mit der gleichzeitigen Möglichkeit für den Autor, den Inhalt etwa auf der eigenen Website zum Download anzubieten.

Sowohl die BOAI als auch die zwei Jahre später folgende *Berliner Erklärung* sind wichtige Meilensteine der Open-Access-Bewegung. In der Präambel der Berliner Erklärung heißt es:

> In Übereinstimmung mit der Budapester Initiative (Budapest Open Access Initiative), der ECHO-Charta und der Bethesda-Erklärung (Bethesda Statement on Open Access Publishing) haben wir diese Berliner Erklärung entworfen, um das Internet als Instrument für eine globale Basis wissenschaftlicher Kenntnisse und geistiger Reflexion zu fördern und um die Maßnahmen zu benennen, die von Politikern, Forschungsorganisationen, Förderinstitutionen, Bibliotheken, Archiven und Museen bedacht werden sollten.
>
> — Berlin Decalaration 2003

Geführte Übungen

1. Können “klassische” Open-Source-Lizenzen auch für Dokumentation und Daten(banken) verwendet werden?

Ja	
Nein	
Das kommt darauf an, es kann keine klare Antwort gegeben werden	

2. Ist es sinnvoll, für Dokumentation separate Lizenzen zu verwenden? Warum?

3. Ist es sinnvoll, für Datenbanken separate Lizenzen zu verwenden? Warum?

4. Welche der folgenden Lizenzen bieten sich für Dokumentation an?

CC-BY-4.0	
FDL	
ODbL	
GPL-3.0	
BSD-3-Clause	

5. Was ist unter dem Begriff “Open Access” zu verstehen?

Offene Übungen

1. Gibt es spezifische Lizenzen für Schriftarten?

2. Erläutern Sie kurz zwei Open-Access-Strategien bzw. -Modelle.

3. Was ist unter der “Open Definition” zu verstehen?

Zusammenfassung

Neben den bekannten Creative-Commons-Lizenzen gibt es weitere Konzepte für die Lizenzierung von Inhalten, die noch spezifischer auf die jeweiligen Datentypen zugeschnitten sind: Für Dokumentation oder Datenbanken stehen mit der FDL oder der ODbL Lizenzen zur Verfügung, die die Eigenschaften dieser Werkarten berücksichtigen. Diese Lektion gibt einen Überblick über einige dieser Lizenzen und geht auch auf das Konzept Open Access ein.

Antworten zu den geführten Übungen

1. Können “klassische” Open-Source-Lizenzen auch für Dokumentation und Daten(banken) verwendet werden?

Ja	X
Nein	
Das kommt darauf an, es kann keine klare Antwort gegeben werden	

2. Ist es sinnvoll, für Dokumentation separate Lizenzen zu verwenden? Warum?

   Ja, weil sie in der Regel andere Inhalte umfassen als das Programm selbst. Zudem regeln sie, wie etwa mit Druckausgaben der Dokumentation umzugehen ist

3. Ist es sinnvoll, für Datenbanken separate Lizenzen zu verwenden? Warum?

   Ja, es kann sinnvoll sein. Insbesondere kann durch eine spezifische Datenbank-Lizenz ein Copyleft-Effekt für die Datenbank erzielt werden, ohne dass sich das Copyleft auf den umgebenden Code auswirkt. Außerdem adressieren Datenbank-Lizenzen spezifische Bestimmungen, die das Urheberrecht für den Schutz von Datenbanken vorsieht.

4. Welche der folgenden Lizenzen bieten sich für Dokumentation an?

CC-BY-4.0	X
FDL	X
ODbL	
GPL-3.0	
BSD-3-Clause	

5. Was ist unter dem Begriff “Open Access” zu verstehen?

   Der Begriff beschreibt den freien Zugang zu wissenschaftlicher Literatur und anderen Inhalten im Internet.

Antworten zu den offenen Übungen

1. Gibt es spezifische Lizenzen für Schriftarten?

   Ja, es gibt zum Beispiel die SIL Open Font License (OFL), die ein Copyleft spezifisch für Fonts enthält, das sich aber nicht auswirkt, wenn Fonts unverändert verwendet werden.

2. Erläutern Sie kurz zwei Open-Access-Strategien bzw. -Modelle.

   Bei der “goldenen” Open-Access-Strategie wird die Veröffentlichung durch den Verlag direkt unter einer Open-Content-Lizenz zur Verfügung gestellt, in der Regel (aber nicht ausschließlich) unter Nutzung von Creative-Commons-Lizenzen.

   Das “grüne” Open-Access-Modell beschreibt, dass zwar die Veröffentlichung in einem Verlag nicht Open Access erfolgt, aber der Autor die Möglichkeit erhält, seine Veröffentlichung zum Beispiel auf seiner eigenen Website unter einer offenen Lizenz zum Download zur Verfügung zu stellen.

3. Was ist unter der “Open Definition” zu verstehen?

   Die “Open Definition” ist eine von der Open Knowledge Foundation formulierte Grundlage für ein gemeinsames Verständnis des Begriffs “Open” in “Open Data”, “Open Knowledge” und “Open Content”. Die Definition beschreibt insbesondere die Grundfreiheiten, die eingeräumt werden müssen, wenn eine Lizenz “Open” im Sinne der Definition sein soll. Dazu gehören unter anderem: der Zugriff, die Nutzung, die Bearbeitung bzw. Veränderung und das Teilen.

Linux Professional Institute

Thema 054: Open-Source-Geschäftsmodelle

054.1 Geschäftsmodelle der Softwareentwicklung

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 054.1

Gewichtung

2

Hauptwissensgebiete

- Verständnis der Ziele und Gründe für die Veröffentlichung von Software oder Inhalten unter einer offenen Lizenz
- Verständnis der gängigen Geschäftsmodelle und Einnahmequellen für Unternehmen, die Open Source Software und offene Inhalte entwickeln
- Verständnis der Auswirkungen der Verwendung von Open Source Software als Komponenten in größeren Technologieprodukten und -dienstleistungen
- Verständnis der Auswirkungen von Lizenzen auf Geschäftsmodelle zur Softwareentwicklung
- Bewusstsein für Kostenstrukturen und Investitionen, die für Geschäftsmodelle zur Entwicklung von Open Source Software erforderlich sind

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Bezahlte Entwicklung
- Open Core und kostenpflichtige Add-ons
- Freemium
- Enterprise- and Community-Versionen
- Self-hosted Distribution
- Subscription

- Kundensupport

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	054 Open-Source-Geschäftsmodelle
Lernziel:	054.1 Geschäftsmodelle der Softwareentwicklung
Lektion:	1 von 1

Einführung

Traditionell hatten Unternehmen, die proprietäre Software (auch bekannt als *Closed Source* Software) vertrieben, ein relativ einfaches Geschäftsmodell: Sie verkauften eine Lizenz zur Nutzung der Software, entweder in Form einer einmaligen Gebühr oder als *Abonnement*.

Doch im Laufe der Jahrzehnte hat sich der Markt für Software radikal verändert: Heute erwarten Kunden, dass sie ein paar Euro für eine App bezahlen und lebenslang kostenlose Updates erhalten.

Infolgedessen ist der herkömmliche Ansatz für proprietäre Software nicht mehr rentabel, und selbst Unternehmen, deren gesamtes Geschäftsmodell auf dem Verkauf von Softwarelizenzen basierte, haben sich auf Integration, Support, Dienstleistungen, Software as a Service (SaaS), Cloud-/Container-Hosting und Hardwareprodukte, auf denen Software läuft (Telefone, Laptops, Server, Drucker, Autos, intelligente Uhren usw.), verlegt.

Im Gegensatz dazu geben Lizenzen für freie und Open Source Software jedem das Recht, die Software zu nutzen, zu studieren, zu verändern und weiterzugeben, ohne dafür eine Gebühr zu

zahlen. Unternehmen, die auf Open Source Software basieren, hatten also nie die Möglichkeit, eine Lizenz für die Nutzung der Software zu verkaufen. Sie sollten also nicht erwarten, dass Sie einfach die Arbeit anderer aus einem Open-Source-Projekt übernehmen und Gewinn damit machen, indem Sie diese öffentlich verfügbare Arbeit Ihren Kunden anbieten; die gleiche Software können Ihre Kunden kostenlos direkt von dem Projekt erhalten. Stattdessen setzen Unternehmen, die freie und Open Source Software anbieten, auf alternative Geschäftsmodelle. Die erfolgreichsten Software-Geschäftsmodelle kombinieren oft mehrere Vorteile für ihre Kunden.

Diese Lektion befasst sich mit der Frage, warum sich ein Unternehmen für Software-Geschäftsmodelle mit Open Source Software entscheiden sollte, und mit den Kompromissen, die sich daraus ergeben.

Ziele und Gründe für die Freigabe von Software oder Inhalten unter einer offenen Lizenz

Es gibt viele Gründe, warum sich Entwickler dafür entscheiden, ein Unternehmen auf Open Source Code aufzubauen: Die Software kann einen Bedarf decken, ein Problem lösen oder Funktionen bereitstellen, die die Entwickler ihren Kunden anbieten. Ein Unternehmen kann Zeit und Geld sparen, indem es die Arbeit an der Software mit anderen Entwicklern teilt, anstatt dieselbe Software von Grund auf neu zu schreiben. Die Teilnahme an dem Projekt ist eine Investition, die sich hoffentlich in nutzbarer, stabiler, zuverlässiger, sicherer und gut gewarteter Software auszahlt.

Viele Open-Source-Entwickler erhoffen sich von ihren Kunden Fehlerkorrekturen (Bugfixes). Einige Kunden fügen sogar neue Funktionen hinzu, portieren die Software auf eine neue Plattform oder nehmen andere bedeutende Verbesserungen vor. Sowohl Fehlerkorrekturen als auch Verbesserungen auf höherem Niveau verbessern die ursprüngliche Software. Wenn Kunden diese an die ursprünglichen Entwickler zurückgeben, können die Änderungen den Code für andere potenzielle Kunden attraktiver machen und seine Verbreitung fördern.

Aber ein Unternehmen kann nicht einfach Beiträge und Aufmerksamkeit erwarten, indem es seinen Code auf einer Website wie GitHub oder GitLab veröffentlicht. Es ist eine Sache, Entwickler zur Teilnahme an einem Projekt zu ermutigen, aber eine größere Herausforderung, sie dafür zu begeistern; unterschätzen Sie also nicht den Arbeitsaufwand, den Community-Beiträge erfordern. Ein Unternehmen muss sich bemühen, Entwickler anzuwerben, sie zu betreuen, sie zu motivieren und ihren Code auf Fehler zu überprüfen.

Ein weiterer Grund für die Öffnung des Codes kann die Schaffung eines Industriestandards sein. Die gemeinsame Nutzung des Codes kann zu besserer Interoperabilität und anderen Vorteilen führen. Das Team, das den Code entwickelt hat, verfügt über wertvolles Fachwissen, das es ihm

ermöglichen könnte, die Zukunft des Projekts zu lenken und für die Unterstützung anderer, die den Code verwenden, bezahlt zu werden.

Einige Unternehmen öffnen ihren Code, um bei Kunden Vertrauen zu schaffen. Erstens wissen die Kunden, dass sie den Code weiter nutzen und verbessern können, wenn das Unternehmen scheitert oder beschließt, in einen anderen Markt einzusteigen und den Code aufzugeben. Zweitens können Kunden Qualität und Sicherheit des Codes selbst überprüfen oder einen unabhängigen Experten mit einem Audit beauftragen, was bei geschlossener, proprietärer Software normalerweise nicht möglich ist.

Schließlich könnte ein Unternehmen eine bereits quelloffene Codebasis übernommen haben, um darauf ein kommerzielles Geschäft aufzubauen. Die Lizenz könnte das Unternehmen dazu verpflichten, den Quellcode seiner Änderungen an jeden Benutzer des Programms weiterzugeben.

Zusammengefasst gibt es also einige Gründe, den Code eines Unternehmens offen zu machen:

- Weiterentwicklung eines bereits bestehenden freien Softwareprojekts
- Beiträge und Innovationen von Kunden
- Kundenvertrauen
- Förderung des Codes als Industriestandard

Geschäftsmodelle und Einnahmequellen

Die Softwareindustrie hat in den vergangenen Jahrzehnten zahlreiche Geschäftsmodelle entwickelt, die sowohl für proprietäre als auch für Open-Source-Unternehmungen geeignet sind. Dieser Abschnitt konzentriert sich auf die beliebtesten Modelle unter Open-Source-Entwicklern.

Ein Geschäftsmodell, das sowohl bei proprietärer als auch bei Open Source Software üblich ist, besteht darin, für Support Geld zu verlangen. Kunden haben möglicherweise Probleme bei Installation und Konfiguration der Software, bei der Behebung von entdeckten Fehlern, beim Hinzufügen neuer Funktionen für den eigenen Einsatz und bei der Verwaltung ihrer Systeme. Mitarbeiter, die die Software entwickelt haben, sind ausgezeichnet für den Support. Tatsächlich bieten diese Mitarbeiter wahrscheinlich bereits kostenlosen Support in Foren an, in denen die Software diskutiert wird.

Beim Support-Modell bezahlt der Kunde den Anbieter dafür, dass er die Open Source Software auf seiner Hardware installiert (Self-Hosting), oder er erhält Zugriff auf die Software auf der Hardware des Anbieters (Cloud-basiertes Modell). Support-Verträge stellen sicher, dass der Kunde rechtzeitig Hilfe vom Unternehmen für seine komplexeren Probleme erhält. Unternehmen, die dieses Modell nutzen, erhalten vom Kunden in der Regel eine regelmäßige (meist monatliche)

Zahlung.

Ein weiteres Geschäftsmodell für proprietäre Software — eigentlich eine Reihe von sich überschneidenden Modellen — nennt sich *Freemium*, ein Begriff, der 2009 vom Autor und Redakteur Chris Anderson eingeführt wurde und auf einem viel älteren Geschäftsmodell basiert, das als *Razor and Blades* (Rasierer und Klingen) bekannt ist. Dabei zahlen Kunden sehr wenig für das Basisprodukt (beispielsweise ein Rasierer oder einen Drucker), aber deutlich mehr für notwendige Komponenten, die sich abnutzen (Klingen für den Rasierer, Tintenpatronen für den Drucker).

Beim Freemium-Modell können Kunden ein Produkt bis zu einem bestimmten Grad kostenlos nutzen und werden dann ermutigt, für weitere Funktionen zu zahlen. Sie kennen wahrscheinlich Online-Nachrichtenseiten, die eine bestimmte Anzahl von Artikeln kostenlos anbieten und die Leser auffordern, ein Abonnement für den vollen Zugang zu ihren Seiten abzuschließen; dies ist ein bekanntes Beispiel für ein Freemium-Modell. Ein anderes Beispiel ist eine Spieleplattform, die das Basisspiel kostenlos zur Verfügung stellt, aber für bestimmte Inhalte Geld verlangt.

Andere proprietäre Softwareunternehmen setzen bei ihrem Freemium-Modell auf Zeit: “Testen Sie die Software drei Monate lang kostenlos und zahlen Sie dann, um sie weiter nutzen zu können.”

Proprietäre Softwareunternehmen nutzen manchmal eine Variante des Freemium-Modells, die als *Open Core* bekannt ist. Dabei sind bestimmte Grundfunktionen (der “Kern” des Produkts) quelloffen, und zusätzliche proprietäre Funktionen können lizenziert werden.

Oft ist der offene Teil voll funktionsfähig, funktioniert aber am besten für Wissenschaftler oder einzelne Benutzer und ist schwer zu verwalten, wenn viele Menschen ihn in einem Unternehmen gemeinsam nutzen. Daher können die proprietären Funktionen bequeme Webschnittstellen für die Verwaltung, Werkzeuge für die Buchhaltung und die Zusammenarbeit und andere Dinge umfassen, die für große Websites von besonderem Interesse sind.

Auch wenn Open-Core-Geschäftsmodelle Open Source Software nutzen, sind Kunden klug genug zu erkennen, dass das gesamte Produkt eigentlich proprietär ist. Open Core mag zwar den Vorteil haben, auf einem bestehenden Open-Source-Projekt aufzubauen, bietet aber nicht die anderen Vorteile von Open Source Software. Ein Open-Core-Geschäftsmodell schafft kein Vertrauen bei Kunden, inspiriert sie nicht dazu, zur Basissoftware beizutragen, gibt Kunden keinen Zugang zur Prüfung des Codes, baut kein Netzwerk von qualifizierten Entwicklern außerhalb des Hauptunternehmens auf und funktioniert nicht als Industriestandard. Schlimmer noch, das Open-Core-Entwicklungsmodell hat die gleichen Entwicklungskosten, ohne die Vorteile, die es lohnenswert machen. Unternehmen, die ein Open-Core-Geschäftsmodell ausprobieren, sind im Allgemeinen von den Ergebnissen enttäuscht, und viele wechseln später zu einem rein

proprietären Modell.

Unternehmen können auch einen Webdienst auf einer quelloffenen oder proprietären Codebasis aufbauen: *Software as a Service* (SaaS). Kunden zahlen auf monatlicher oder jährlicher Basis.

Viele Unternehmen, die SaaS betreiben oder mobile Apps anbieten, verdienen ihr Geld mit Werbung. Einige Unternehmen sammeln auch Daten über die Benutzer des Dienstes oder der App und verkaufen diese Daten an Dritte, die sie für Werbung nutzen können. Werbung kann jedoch als lästig empfunden werden. Der Verkauf von Daten ist umstritten, und einige Länder schränken die Datenerfassung ein.

Schließlich können Unternehmen auch Open Source Software entwickeln und veröffentlichen, ohne zu versuchen, damit Geld zu verdienen. Dieses Modell steht Unternehmen zur Verfügung, die nicht nur mit der Software Geld verdienen, sondern auch mit anderen Dingen. So arbeiten beispielsweise Automobilhersteller zusammen, um große Mengen an freier Software für ihre Autos zu entwickeln (die heutzutage vollständig computerisiert sind).

Große Softwareunternehmen, die ihre Umsätze mit proprietären Angeboten erzielen, wie etwa Google und Amazon, geben manchmal Verwaltungssoftware oder andere nützliche Tools als Open Source frei, da diese Tools nicht zu ihrem Kerngeschäft gehören. Die Unternehmen, die den Code unter einer offenen Lizenz freigeben, profitieren von den Rückmeldungen, Fehlerberichten und Funktionsverbesserungen der Open Source Community.

Open Source Software in anderen Technologien und Diensten

Viele Unternehmen integrieren Open Source Software in ihre Produkte oder Webplattformen; schließlich ist Open Source Software kostenlos. Aber das ist nicht der wichtigste Grund (und vielleicht nicht einmal ein guter Grund), sie zu übernehmen. Viel wichtiger ist, dass diese Software meist von hoher Qualität ist (obwohl das Entwicklerteam sie sorgfältig prüfen sollte, bevor es sie übernimmt) und vielleicht sogar ein Industriestandard ist.

Ein weiterer Vorteil von Open Source Software besteht darin, dass sie auch dann noch verfügbar ist, wenn die Entwickler oder die Gemeinschaft, die sich um sie herum gebildet hat, verschwinden.

Aber freie und Open Source Software bringt auch Verantwortung mit sich. In diesem Abschnitt werden kurz die wichtigsten Punkte aufgeführt, die vor der Übernahme zu beachten sind.

Die ersten Fragen gelten für jede Software von Drittanbietern, die für eine Übernahme in Frage kommt: Wird die Software den Bedürfnissen der Benutzer jetzt und in Zukunft gerecht? Wird die Software von einer starken Gemeinschaft unterstützt, die technischen Support bieten und die Software weiterentwickeln kann? Weist die Software erhebliche Sicherheitslücken auf?

Darüber hinaus müssen Manager die Verantwortung des Unternehmens im Blick haben, wenn es ein Open-Source-Projekt in seine eigene Software einbindet. Wenn Entwickler neuen Code auf dem Open Source Code aufbauen, ist zu prüfen, was die Lizenz der Open Source Software vorgibt. Einige Lizenzen verlangen, dass der Quellcode der Änderungen unter der gleichen freien Lizenz wie die ursprüngliche Codebasis weitergegeben wird.

Selbst wenn ein Entwickler nicht verpflichtet ist, veränderten Quellcode weiterzugeben, und ein proprietäres Produkt auf der Grundlage des Open Source Codes erstellt, möchte er vielleicht bestimmte Dinge zurückgeben, wie beispielsweise Fehlerkorrekturen und Erweiterungen des Open Source Codes. Indem er diese Korrekturen und Erweiterungen beisteuert, erlaubt er dem Projekt, sie zu übernehmen (wenn die Kernentwickler sich dazu entschließen) und zu pflegen. Entwickler, die keine Verbesserungen zurückgeben, müssen die Korrekturen und Erweiterungen möglicherweise jedes Mal neu implementieren, wenn sie auf eine neue Version des ursprünglichen Codes aktualisieren.

Wegen dieser Abhängigkeit und aus anderen Gründen sollten die Mitarbeiter des Unternehmens in Erwägung ziehen, aktive Mitglieder der Community zu werden, die den Code entwickelt. Die Entwickler können durch ihre Mitarbeit viel über den Code lernen und die Richtung der zukünftigen Entwicklung mitbestimmen. Natürlich sollte das Unternehmen seine Entwickler für die Zeit bezahlen, die sie in die Aktivitäten der Community investieren. Für ein Unternehmen, das freie Software ernsthaft nutzen möchte, ist sie nicht kostenlos.

Open Source Software aus Sicht des Kunden

Die Öffnung des Codes ist für Kunden aus mehreren Gründen von Vorteil, setzt aber auch eine andere Beziehung zwischen dem Unternehmen und seinen Kunden voraus. Die Kunden sollten sowohl die Vorteile als auch die Implikationen verstehen.

Der bereits erwähnte Hauptvorteil für Kunden besteht darin, dass sie größeres Vertrauen in die Software haben können. Sie wissen, dass die Software nicht verschwinden wird. Viele Unternehmen schließen oder schlagen plötzlich eine neue Richtung ein und lassen ihre Kunden im Stich, die nur eine kurze Frist haben, um auf ein anderes Produkt umzusteigen, das ihnen vielleicht gar nicht gefällt. Open Source Software hingegen hängt nicht von einer einzelnen Organisation ab. Wenn das Projekt wichtig ist, werden es andere Mitglieder der Community weiterführen, wenn sich die ursprünglichen Entwickler zurückziehen.

Kunden können zudem Qualität und Sicherheit von Open Source Software prüfen und testen, wie leicht sie eigene Funktionen hinzufügen oder die Software auf eine neue Umgebung portieren können.

Da der Quellcode eines Open-Source-Projekts für alle einsehbar ist, kann sich ein breites

Spektrum von Entwicklern damit vertraut machen. Bei einem Projekt mit aktiver Community bieten viele Mitglieder im Forum Unterstützung an. Oft ist es einfach, Leute für Support, Verwaltung und Nutzung der Software zu finden, da neue Mitarbeiter den Code bereits kennen.

Es ist sehr beruhigend für Kunden, die in ihrem Tagesgeschäft auf den Code angewiesen sind, dass sie Fehler selbst beheben oder jemanden damit beauftragen können. Bei einem obskuren Fehler, der nur wenige Kunden betrifft, kann es Jahre dauern, bis er vom Hauptentwicklungsteam proprietärer Software behoben wird, was für Benutzer permanente Frustration bedeutet. Ist der Quellcode verfügbar, sind Fehler schneller zu erkennen und zu beheben.

Was bedeutet das für die Beziehung zwischen einem Unternehmen und seinen Kunden? Ein Unternehmen kann sich für ein typisches proprietäres Modell entscheiden, indem es seinen Kunden regelmäßige Updates und einen Supportvertrag anbietet. Der Kunde muss sich niemals den Quellcode ansehen oder den Community-Foren beitreten.

Die meisten Kunden würden jedoch von den einzigartigen Möglichkeiten profitieren, die Open-Source-Projekte bieten: Sie können ihren eigenen Entwicklern erlauben, sowohl Informationen als auch Code beizusteuern. Eine solche Beteiligung vertieft das Verständnis der Mitarbeiter und hilft, neue Mitarbeiter zu rekrutieren. Zudem eröffnet es die Möglichkeit, die zukünftige Entwicklung mitzugestalten.

Kostenstrukturen und Investitionen

Programmierer sind sehr gefragt, so dass jede Softwareentwicklung kostenintensiv ist. Experten für große Open-Source-Projekte sind sogar noch gefragter, weil die jeweilige Codebasis weit verbreitet sind.

Ein Open-Source-Projekt bietet Potenzial für Kosteneinsparungen, da verschiedene Organisationen und Einzelpersonen ihre Bemühungen in einer gemeinsamen Codebasis bündeln können. Der Betrieb eines solchen Projekts führt jedoch zu neuen Kosten.

Wenn ein Unternehmen ein Projekt öffnet, das intern entwickelt wurde, muss es investieren, um es für eine größere (möglicherweise weltweite) Community nutzbar zu machen. Funktionen, die auf die geschäftlichen Anforderungen des Unternehmens zugeschnitten waren, müssen möglicherweise überdacht werden, um auch anderen Unternehmen zu nutzen.

Ist der Code mangelhaft oder umständlich, sollte ein Unternehmen ihn wahrscheinlich nicht freigeben. Die Qualität könnte potenzielle Beiträger oder auch Kunden abschrecken. Es liegt jedoch im Interesse der Entwickler, diese Probleme trotzdem zu beheben, denn schlecht programmierte Software ist sperrig: Sie lässt sich nur schwer mit neuen Funktionen aktualisieren und neigt dazu, komplexe Fehler zu werfen, die schwer zu beheben sind. Diese Probleme werden

technical debt (technische Schulden) genannt, und je früher sie behoben werden, desto besser ist es für alle Beteiligten.

Schließlich ist es erstaunlich, wie oft Unternehmen Geschäftsgeheimnisse, Passwörter, persönliche Hinweise auf Personen oder andere sensible Informationen in Code einbetten. Entwickler müssen Zeit investieren, um solche Schwachstellen zu beseitigen. Passwörter, API-Schlüssel, Zertifikate und Cloud-Zugangsdaten sollten nie im Code stehen, sondern extern über einen sicheren Dienst verwaltet werden.

Nehmen wir an, ein Unternehmen hat seinen Code freigegeben und hofft, von externen Beiträgen zu profitieren. Einige Kunden werden Entwickler für die Wartung und neue Funktionen bezahlen. Andere werden ihre eigenen Entwickler auf das Projekt ansetzen, aber das Hauptentwicklungsteam muss Zeit für deren Unterstützung aufwenden. Das Hauptentwicklungsteam muss Außenstehende über den Code und die damit verbundenen Codierungsstandards aufklären. Dieses Team sollte Außenstehende auch anleiten, was sie hinzufügen und wohin sie Kommentare zu ihrem Code zurücksenden sollen.

Halten Sie Ausschau nach externen Helfern, die sich als talentiert erweisen. Sie könnten wertvolle Verstärkung für das Kernteam sein. Sie wollen sich möglicherweise gern dem Team anschließen und beträchtliches Wissen einbringen.

Wenn ein Team von Entwicklern bezahlt wird, während andere ihre Zeit freiwillig zur Verfügung stellen, könnten sich die Freiwilligen ausgenutzt fühlen oder sich fragen, warum sie helfen sollten. Jeder, der einen Beitrag leistet, sei es ein einzelner Freiwilliger oder eine Organisation, muss die zuvor in dieser Lektion beschriebenen Überlegungen anstellen, um zu entscheiden, ob ein Beitrag sinnvoll ist.

Freier Code ist nicht kostenlos, auch wenn er nicht mit Lizenzkosten verbunden ist. Es handelt sich schlichtweg um ein anderes Entwicklungsmodell.

Geführte Übungen

1. Wie kann Open Source Code das Vertrauen von Kunden in die Software steigern?

2. Was könnte Unternehmen veranlassen, ein Open-Core-Geschäftsmodell zu versuchen, und warum kann dieses Modell die Vorteile von Open Source Software nicht nutzen?

3. Was sind typische Formen der Unterstützung, die Entwickler von Open-Source-Projekten externen Mitwirkenden zukommen lassen?

Offene Übungen

1. Sie erwägen, ein proprietäres Produkt auf der Basis eines Open-Source-Projekts zu entwickeln. Welche Faktoren müssen Sie bei der Entscheidungsfindung berücksichtigen?

2. Sie haben auf der Grundlage eines Open-Source-Projekts mehrere Produkte entwickelt, die Sie im Abonnement vertreiben. Was werden Sie tun, wenn die Open-Source-Lizenz vorschreibt, dass Sie Ihren gesamten Code wieder in das Projekt einbringen müssen? Außerdem haben Sie dem Open-Source-Projekt Unterstützung für einige neue Kommunikationsprotokolle hinzugefügt, um Ihre eigenen Anforderungen zu erfüllen. Wenn Sie die Wahl haben, werden Sie das Open-Source-Projekt bitten, diese Protokollunterstützung in seinen Kerncode zu integrieren?

Zusammenfassung

In dieser Lektion haben Sie Vorteile von der Öffnung von Quellcode kennengelernt. Sie haben sich mit den heute üblichen Geschäftsmodellen befasst und erfahren, wie wichtig es ist, die Auswirkungen der Lizenz des Codes zu verstehen. Sie haben gesehen, was bei der Einbindung von Open Source Code in Ihr Unternehmen zu beachten ist und wie Ihre Kunden von Open Source profitieren. Außerdem haben Sie gelernt, wie sich die Kosten der Entwicklung von Open Source Software von der proprietärer Software unterscheiden.

Antworten zu den geführten Übungen

1. Wie kann Open Source Code das Vertrauen von Kunden in die Software steigern?

   Kunden können Qualität und Sicherheit des Codes überprüfen. Sie können auch darauf vertrauen, dass sie den Code weiter verwenden können, wenn die ursprünglichen Entwickler den Support einstellen.

2. Was könnte Unternehmen veranlassen, ein Open-Core-Geschäftsmodell zu versuchen, und warum kann dieses Modell die Vorteile von Open Source Software nicht nutzen?

   Auf den ersten Blick scheint Open Core alle Vorteile von Open Source Software zu bieten und gleichzeitig einem Unternehmen die Möglichkeit für ein proprietäres Geschäftsmodell zu geben, indem es Lizenzen für die Nutzung der Software verkauft. In der Praxis bietet ein Open-Core-Geschäftsmodell jedoch nicht die Vorteile von Open Source und hat gleichzeitig die höheren Kosten einer offenen Entwicklung.

3. Was sind typische Formen der Unterstützung, die Entwickler von Open-Source-Projekten externen Mitwirkenden zukommen lassen?

   Die Entwickler eines Projekts leiten in der Regel externe Beiträger an, betreuen sie und überprüfen, ob deren Beiträge den Qualitäts- und Codierungsstandards des Teams entsprechen.

Antworten zu den offenen Übungen

1. Sie erwägen, ein proprietäres Produkt auf der Basis eines Open-Source-Projekts zu entwickeln. Welche Faktoren müssen Sie bei der Entscheidungsfindung berücksichtigen?

   Entscheiden Sie zunächst, ob die Open Source Software Ihren Anforderungen entspricht. Prüfen Sie die Qualität, öffentlich gemeldete Sicherheitsmängel und den Zustand der Community. Prüfen Sie die Lizenz sorgfältig, um festzustellen, ob Sie Ihre Änderungen am Code weitergeben müssen. Legen Sie fest, inwieweit Sie Teil der Community des Open-Source-Projekts sein wollen und welche Rolle Ihre Entwickler in dieser Community spielen werden.

2. Sie haben auf der Grundlage eines Open-Source-Projekts mehrere Produkte entwickelt, die Sie im Abonnement vertreiben. Was werden Sie tun, wenn die Open-Source-Lizenz vorschreibt, dass Sie Ihren gesamten Code wieder in das Projekt einbringen müssen? Außerdem haben Sie dem Open-Source-Projekt Unterstützung für einige neue Kommunikationsprotokolle hinzugefügt, um Ihre eigenen Anforderungen zu erfüllen. Wenn Sie die Wahl haben, werden Sie das Open-Source-Projekt bitten, diese Protokollunterstützung in seinen Kerncode zu integrieren?

   Wenn das Projekt nicht verlangt, dass Sie Änderungen am Code weitergeben, werden Sie dies wahrscheinlich nicht tun, da Sie ein proprietäres Produkt leichter im Abonnement vertreiben können. Wenn Sie Ihren Code offenlegen müssen, bieten Sie ein Abonnement für eine selbst gehostete Distribution an. Auch wenn Sie Ihren Code nicht weitergeben müssen, sollten Sie das Projekt bitten, Ihre Unterstützung für die Kommunikationsprotokolle zu integrieren, damit Sie diese Unterstützung nicht jedes Mal neu implementieren müssen, wenn Sie eine neue Version des Open Source Codes installieren.

054.2 Geschäftsmodelle für Serviceprovider

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 054.2

Gewichtung

2

Hauptwissensgebiete

- Verständnis der gängigen Geschäftsmodelle und Einnahmequellen für Organisationen, die Dienstleistungen im Zusammenhang mit Open Source Software und offenen Inhalten anbieten
- Verständnis der Auswirkungen von Lizenzen auf die Geschäftsmodelle von Dienstleistern
- Verständnis von Service Level Objectives und Service Level Agreements
- Verständnis für die Notwendigkeit des Schutzes von Sicherheit und Privatsphäre
- Kenntnis der Kostenstrukturen und Investitionen, die für Geschäftsmodelle für Open Source Softwaredienste erforderlich sind

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Hosted Services
- Clouds
- Consulting
- Training
- Hardwareverkauf
- Kundensupport
- Terms of Service (ToS)

- Service Level Objectives (SLO)
- Service Level Agreements (SLA)
- Data Processing Agreements

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	054 Open-Source-Geschäftsmodelle
Lernziel:	054.2 Geschäftsmodelle für Serviceprovider
Lektion:	1 von 1

Einführung

Serviceprovider sind das Rückgrat eines jeden Unternehmens oder einer Organisation, die in irgendeiner Weise über ein Netzwerk arbeitet. Vom Internet Service Provider (ISP), der den Zugang zum weltweiten Internet bereitstellt, bis hin zu spezialisierten Anwendungen wie E-Mail oder Kundenmanagement sind es softwarebasierte Dienste, die den eigentlichen Geschäftsbetrieb ermöglichen.

Open Source Software ist Bestandteil vieler verschiedener Geschäftsmodelle von Serviceprovidern. Ein Open-Source-Betriebssystem könnte beispielsweise die Grundlage eines Hosting-Dienstes bilden. Ein Open-Source-Orchestrierungs-Tool könnte die Basis einer Unternehmung im Bereich Infrastructure-as-a-Service (IaaS) oder Platform-as-a-Service (PaaS) sein, wie beispielsweise eines Public-Cloud-Anbieters oder einer Container-Plattform. Eine Open-Source-Anwendung könnte online als Software-as-a-Service (SaaS) bereitgestellt werden. Einige Serviceprovider bieten auch zusätzliche Dienstleistungen im Zusammenhang mit Open Source Software an, wie etwa Beratung, Schulung oder Kundensupport.

Einnahmequellen

Da Open Source Software heruntergeladen und genutzt werden kann, ohne dass dafür eine Gebühr anfällt, haben sich im Rahmen von Open-Source-Geschäftsmodellen spezifische Produkte und Dienstleistungen entwickelt, die unter anderem davon abhängen, wie hoch der Anteil an freier Software im eigentlichen Produkt ist.

Steht die gesamte Software unter einer freien Lizenz, kann ein Geschäftsmodell im *Hosting* bestehen, also in der Bereitstellung der Software auf einer zuverlässigen und sicheren Plattform. Damit entfallen für den Kunden Aufwand und Risiko beim Betrieb eines eigenen Servers. Im Gegenzug zahlt der Kunde dem Serviceprovider Gebühren für die Nutzung. Die Gebühren können sich nach der Anzahl der Benutzerkonten, dem gespeicherten oder übertragenen Datenvolumen oder nach bestimmten Zeitintervallen richten. In manchen Fällen bieten verschiedene Anbieter kostenpflichtige Dienste für dieselbe Open Source Software an.

Bei diesem beliebten Geschäftsmodell handelt es sich häufig um *Abonnements*, bei denen die Kunden eine monatliche oder jährliche Gebühr für den Zugang zu dem Dienst zahlen. Der Zugang wird häufig mit zusätzlichen Diensten oder Funktionen kombiniert. Die Anbieter können mehrere Stufen von Abonnements definieren, bei denen sie für mehr Funktionen, mehr Benutzer, umfangreichere Dienste oder eine höhere Zuverlässigkeitsgarantie einen höhereren Preis verlangen. Manche Provider bieten eine Grundstufe des Dienstes auch kostenlos an, eine Variante des “Freemium”-Modells.

Eine weitere wichtige Einnahmequelle für das Geschäftsmodell von Open Source Serviceprovidern ist seit jeher der *Support*. Dieser umfasst Beratung, Dokumentation und Schulung für Benutzer, die Unterstützung beim Einsatz der Software benötigen. Wenn der Kunde es vorzieht, die freie Software auf seinen eigenen Servern zu hosten, kann sich insbesondere die Installation als schwierig erweisen. Möglicherweise sind unterstützende Tools und Bibliotheken zu installieren, und der Benutzer weiß nicht, wo er diese ablegen muss, damit alle Teile zusammen funktionieren; oder er stößt auf seiner speziellen Hardware und seinem Betriebssystem auf Probleme. Daher sind Support und Schulung durch einen Serviceprovider wertvoll.

Auch *Zusatzleistungen*, wie beispielsweise ergänzende funktionale oder sicherheitsrelevante Features, können kundenspezifisch entwickelt und angeboten werden. Solche Features oder Anpassungen der Software an die ganz spezifischen Anforderungen eines Kunden sind eine anspruchsvolle und lukrative Ergänzung des Geschäftsmodells. Es liegt dann im Ermessen des Dienstleisters und des Kunden, ob die Ergänzungen als freie Software in das Basisprojekt zurückfließen oder proprietär werden.

Einige Open-Source-Projekte bieten ein *Doppellizenzmodell* an: Die Software ist unter einer freien

Softwarelizenz verfügbar, was den Bedürfnissen vieler Benutzer entgegenkommt. Einige Benutzer möchten die Software jedoch in ein proprietäres Produkt einbinden, was unter einer reziproken Lizenz nicht möglich ist. Daher wird für diese Benutzer eine auf dem Urheberrecht basierende Standardlizenz bereitgestellt. Das Doppellizenzmodell ist am effektivsten für Datenbanken, da viele Unternehmen eine funktionsreiche und effiziente Datenbank als Teil eines proprietären Softwareprodukts anbieten möchten.

Bei einem *Werbeeinnahmenmodell* zahlen Kunden nicht für den Zugang zum Dienst. Stattdessen zeigt der Serviceprovider Kunden, die den Dienst nutzen, Werbung an und verlangt von den Unternehmen Gebühren für die Platzierung ihrer Werbung. Open Source Serviceprovider vermeiden in der Regel proprietäre Werbedienste aufgrund von Bedenken hinsichtlich des Datenschutzes ihrer Benutzer. Alternative Open-Source-Werbedienste führen jedoch kein invasives Ad-Tracking durch.

Bei einem *provisionsbasierten Modell* erhält der Serviceprovider einen Prozentsatz oder eine Gebühr für die Verwaltung von Transaktionen über seinen Onlinedienst. Dieses Modell ist die Grundlage für die meisten E-Commerce-Websites, Reisebuchungsseiten und Zahlungsabwickler. Viele Open-Source-Projekte in diesem Bereich sind Allzweckplattformen, die entweder vom Dienstanbieter selbst gehostet oder als sekundärer provisionsbasierter Dienst angeboten werden, bei dem die kundenseitige Website einen Prozentsatz oder eine Gebühr für eine gehostete Version der Open-Source-Plattform zahlt.

Ein *Open-Content-Geschäftsmodell* umfasst die Erstellung und Verbreitung von Inhalten unter Lizenzen wie Creative Commons, so dass andere sie frei nutzen, verändern und weitergeben können. Die Benutzer werden ermutigt, zu den Inhalten beizutragen und damit im Laufe der Zeit zu verbessern. Während die Inhalte selbst kostenlos sind, können Einnahmen durch ergänzende Produkte oder Dienstleistungen erzielt werden, beispielsweise durch Spenden, die Anzeige von Werbung, das Angebot von Freemium-Funktionen, den Verkauf von zugehörigen Waren oder die Bereitstellung von Beratungs-, Schulungs- oder Supportleistungen.

Die Wahl zwischen gehosteten und selbst gehosteten Diensten

Gehostete Dienste sind ideal für Unternehmen, die Benutzerfreundlichkeit, niedrige Einstiegskosten, Skalierbarkeit und ein professionelles Sicherheitsmanagement ohne eigene tiefgreifende technische Fachkenntnisse wünschen. Diese Dienste haben jedoch auch potenzielle Nachteile in Bezug auf Kontrolle, Anpassung, Datenschutz und Abhängigkeit vom Anbieter.

Gehostete Dienste erfordern in der Regel nur ein minimales Setup, und die Serviceprovider kümmern sich um Wartung, Aktualisierungen und Systemsicherheit. Sie stellen oft globale Rechenzentren zur Verfügung, die höhere Leistung und größere Zuverlässigkeit für internationale Benutzer bieten. Serviceprovider können Ressourcen je nach Bedarf mit minimalem Aufwand für

den Kunden aufstocken oder verringern. Außerdem haben gehostete Dienste oft spezielle Sicherheitsteams, die Bedrohungen erkennen und darauf reagieren; darüber hinaus sorgen Serviceprovider für die Einhaltung verschiedener gesetzlicher Standards.

Andererseits werden bei gehosteten Diensten sensible Kundendaten extern gespeichert, was hinsichtlich Datenschutz und Vertrauen zu bedenken ist. Eine gemeinsam genutzte Hosting-Umgebung kann zusätzliche Sicherheitsrisiken bergen. Ausfälle oder Ausfallzeiten des Serviceproviders wirken sich direkt auf die Verfügbarkeit der Dienste für den Kunden aus. Die Abhängigkeit von einem einzigen Anbieter kann problematisch sein, wenn dieser die Bedingungen ändert oder den Dienst einstellt.

Selbst gehostete Dienste eignen sich daher für Unternehmen, die über das nötige technische Know-how und die Ressourcen für die Verwaltung ihrer Infrastruktur verfügen. Selbst gehostete Dienste sind jedoch mit höheren Initialkosten, laufender Wartung und Herausforderungen bei der Skalierbarkeit verbunden. Die Entscheidung zwischen gehosteten und selbst gehosteten Diensten hängt letztendlich von den Bedürfnissen, den technischen Möglichkeiten, dem Budget und den Prioritäten in Bezug auf Kontrolle, Anpassung und Datenschutz des jeweiligen Kunden ab.

Auswirkungen von Lizenzen

Grundsätzlich passen Open-Source-Softwarelizenzen gut zu den Geschäftsmodellen von Serviceprovidern, weil der Kunde nicht erwartet, die Software zu besitzen, die den Dienst bereitstellt, sondern für den Zugang zu diesem Dienst zahlt.

Andererseits wirft der Einsatz von Open Source Software in Diensten wie IaaS, SaaS oder PaaS rechtliche Fragen auf, die oft nicht geklärt sind. Nur wenige Lizenzen, insbesondere die *GNU Affero General Public License*, regeln ausdrücklich die Verwendung von freier Software “im Falle von Netzwerkserver-Software”.

Bei anderen verbreiteten Lizenzen — sowohl bei den restriktiveren Copyleft-Lizenzen wie der GNU General Public License als auch bei permissiven Lizenzen wie den BSD-Lizenzen — gibt es Ermessensspielräume, was bei der Nutzung der Software über das Netz als “kopieren”, “überlassen” oder “weitergeben” anzusehen ist. Von solchen Definitionen hängen Entscheidungen darüber ab, ob und in welcher Form der Quellcode der Anwendung bereitgestellt werden muss oder welche Anforderungen an die Namensnennung gestellt werden. Auch die Verbindung mit proprietären Softwarekomponenten oder die Lizenzierung von selbst entwickelten Zusatzkomponenten muss rechtlich geklärt werden. Aus diesen Gründen ist es im Rahmen von Geschäftsmodellen von Serviceprovidern von entscheidender Bedeutung, lizenzrechtliche Fragen im Zusammenhang mit der Nutzung von freier Software zu klären.

Es ist gute Praxis, dass Serviceprovider sämtliche von ihnen eingesetzte Open Source Software zusammen mit der jeweiligen Lizenz öffentlich auflisten, auch wenn sie dazu nicht gesetzlich verpflichtet sind.

Überlegungen zur Sicherheit und zum Schutz der Privatsphäre

Der Erfolg eines Serviceproviders hängt in hohem Maße von der Kundenzufriedenheit ab, insbesondere bei Open Source Software, bei der die Kunden die Freiheit haben, die Software selbst zu hosten oder einen konkurrierenden Anbieter für dieselbe Software zu wählen. Für die Kunden sollte es bequemer, kostengünstiger, zuverlässiger, sicherer oder leistungsfähiger sein als das Self-Hosting. Der Nachteil besteht jedoch darin, dass Kunden ihre Daten mit dem Anbieter austauschen müssen, was zu Bedenken hinsichtlich des Datenschutzes bei persönlichen oder sensiblen Daten führen kann.

Im Hinblick auf Sicherheit sollten sich Kunden vergewissern, dass der Serviceprovider einen verlässlichen Prozess für die zeitnahe Anwendung von Sicherheitspatches hat. Es lohnt sich zu prüfen, wie der Dienst Abhängigkeiten von anderen Open-Source-Projekten verwaltet, um sicherzustellen, dass die gesamte Software auf dem neuesten Stand und sicher ist. Der Kunde sollte auch die Sicherheitsrichtlinien des Anbieters bewerten, einschließlich der Handhabung von Datenverschlüsselung, Zugriffskontrollen und der Reaktion auf Sicherheitszwischenfälle.

Die Transparenz von Open Source Software ermöglicht eine gründliche Überprüfung des Codes durch die Community, was die Sicherheit durch die Identifizierung und Behebung von Schwachstellen erhöhen kann. Kunden sollten daher auf Überprüfungen oder Audits der Software durch seriöse Dritte oder Sicherheitsexperten in der Community achten. Diese Audits sollten bestätigen, dass der Provider im Entwicklungsprozess sichere Codierungspraktiken und -standards anwendet, und sie sollten berücksichtigen, ob der Provider CI/CD-Pipelines mit integrierten Sicherheitsprüfungen nutzt, um Schwachstellen frühzeitig zu erkennen.

Im Hinblick auf den Datenschutz sollten Kunden sicherstellen, dass der Dienst nur die für seinen Betrieb erforderlichen Daten erfasst und speichert, um das Risiko der Preisgabe von Daten zu minimieren. Der Dienst sollte die Daten sowohl bei der Übertragung als auch bei der Speicherung mit einer starken Verschlüsselung versehen. Mechanismen zur Zugangskontrolle sollten sicherstellen, dass nur befugtes Personal Zugang zu sensiblen Daten erhält.

Der Kunde sollte sich vergewissern, dass der Dienst den Benutzern die Kontrolle über ihre Daten gibt, beispielsweise die Möglichkeit, sie zu löschen oder zu exportieren. Der Dienst sollte die einschlägigen Datenschutzbestimmungen einhalten — etwa die General Data Protection Regulation (GDPR) der Europäischen Union und den California Consumer Privacy Act (CCPA) — und Transparenz über seine Praktiken zur Datenverarbeitung bieten.

Der Kunde sollte auch die Datenschutzrichtlinien des Anbieters prüfen, um zu verstehen, wie Daten gesammelt, verwendet, weitergegeben und geschützt werden, und um sicherzustellen, dass Dritte, mit denen der Anbieter Daten austauscht, ebenfalls strenge Datenschutzstandards einhalten.

Im Allgemeinen ist es sinnvoll, das Feedback und die Bewertungen der Community zu lesen, um den Ruf und die Vertrauenswürdigkeit der Software und des Serviceproviders zu beurteilen. Eine solide Community oder Organisation sollte das Open-Source-Projekt aktiv pflegen und unterstützen. Der Kunde sollte sich vergewissern, dass der Anbieter über regelmäßige Datensicherungsverfahren und solide Notfallwiederherstellungspläne verfügt, und prüfen, ob der Dienst Datenredundanz nutzt, um Verfügbarkeit und Zuverlässigkeit zu gewährleisten.

Serviceprovider sollten sich darüber im Klaren sein, dass ihre Kunden ihre Sicherheits- und Datenschutzpraktiken, ihren Ruf in der Community, die Einhaltung von Vorschriften und die Transparenz im Umgang mit Daten bewerten, und sollten daher auf bewährte Verfahren setzen.

Vereinbarungen zwischen Serviceprovider und Kunden

Serviceprovider schließen in der Regel Verträge, die die rechtliche und operative Grundlage der Beziehung zwischen dem Unternehmen und seinen Kunden bilden. Diese Vereinbarungen gewährleisten klare Kommunikation, legen Erwartungen fest, definieren Verantwortlichkeiten und sichern beide Parteien rechtlich ab.

Die *Terms of Service* (ToS) für einen Dienst, auch bekannt als *Terms and Conditions* (T&C) oder *Terms of Use* — also die Geschäfts- oder Nutzungsbedingungen — sind eine rechtlich bindende Vereinbarung zwischen einem Serviceprovider und den Kunden oder Benutzern des Dienstes. Die ToS umreißen die Regeln, Verantwortlichkeiten und Beschränkungen, denen die Nutzung des Dienstes unterliegen. Sie schützen sowohl den Provider als auch den Kunden, indem sie klar definieren, was jede Partei bei der Bereitstellung oder Nutzung des Dienstes darf und was nicht. Einige übliche Punkte einer ToS-Vereinbarung sind die Bestätigung, dass der Benutzer mit der Einhaltung der Bedingungen einverstanden ist, Richtlinien zu angemessenem Verhalten und verbotenen Aktivitäten, Einzelheiten in Bezug darauf, wem die von den Benutzern erstellten oder hochgeladenen Inhalte gehören, Informationen über Einrichtung, Sicherheit und Löschung von Benutzerkonten, Schlichtungs- oder Mediationsverfahren zur Lösung von Konflikten und Haftungsbeschränkungen des Providers bei Schäden.

In einer *Privacy Policy* (Datenschutzbestimmungen) ist dargelegt, wie der Anbieter Benutzerdaten sammelt, verwendet, speichert und schützt. Übliche Punkte in Datenschutzbestimmungen sind die Arten von Daten, die der Provider sammelt, ihre Erfassungsmethoden, die Art und Weise, wie er die Daten verwendet, die Bedingungen, unter denen der Provider Daten an Dritte weitergeben darf, und Informationen über die Rechte der Benutzer in Bezug auf ihre Daten, wie beispielsweise

Zugriff, Bearbeitung und Löschung.

Ein *Service Level Agreement* (SLA), also eine Dienstleistungsvereinbarung, ist eine formelle, dokumentierte Vereinbarung zwischen einem Serviceprovider und einem Kunden, in der der erwartete Service-Umfang einschließlich der spezifischen Leistungsmetriken, Verantwortlichkeiten und Erwartungen festgelegt wird. Das SLA definiert die Standards für die Bereitstellung des Service und bietet einen klaren Rahmen für die Messung und Verwaltung der Leistungen. Zu den üblichen Punkten eines SLA gehören eine detaillierte Beschreibung der bereitgestellten Dienste, spezifische Ziele der Leistung, wie etwa Betriebs- und Reaktionszeiten, Methoden für die Leistungsmessung und -berichte, Konsequenzen bei Nichterfüllung der Leistungsziele sowie Verfahren zur Überprüfung und Aktualisierung des SLA.

Ein *Service Level Objective* (SLO), also eine Zielsetzung für einen Dienst, ist eine Schlüsselkomponente eines Service Level Agreements, das messbare Zielvorgaben für die Serviceleistung festlegt. Diese Zielvorgaben quantifizieren den erwarteten Servicelevel zwischen einem Serviceprovider und einem Kunden und stellen sicher, dass der Service die vereinbarten Standards konsequent erfüllt. Zu den üblichen Punkten von SLOs gehören spezifische Messgrößen, die der Serviceprovider zur Bewertung der Serviceleistung misst (etwa Betriebszeit, Antwortzeit, Lösungszeit und Durchsatz), die gewünschten oder erwarteten Werte für jede Leistungsmessgröße, die Techniken und Tools, die zur Messung und Überwachung der Leistungsmessgrößen verwendet werden, die spezifischen Komponenten oder Aspekte des Service, die das SLO abdeckt (beispielsweise Hardware, Software, Netzwerkkomponenten oder spezifische Prozesse), und die Konsequenzen, falls der Service die SLOs (nicht) erfüllt (beispielsweise finanzielle Strafen, Servicegutschriften oder Leistungsboni).

Ein *Data Processing Agreement* (DPA), also eine Vereinbarung zur Datenverarbeitung, ist ein Vertrag zwischen einem Kunden (dem Datenverantwortlichen) und einem Serviceprovider (dem Datenverarbeiter), der an der Verarbeitung personenbezogener Daten beteiligt ist. In der DPA werden die Zuständigkeiten und Pflichten beider Parteien im Hinblick auf die Einhaltung der Datenschutzgesetze, wie etwa der DSGVO, im Einzelnen festgelegt. In einigen Rechtsordnungen ist eine DPA zwischen einem Dienstleister und einem Kunden obligatorisch.

Kostenstrukturen und Investitionen

Die wichtigsten Ausgaben in einem Geschäftsmodell für Serviceprovider sind die Fixkosten im Zusammenhang mit dem Hosting der Dienste, wie etwa der Kauf von Serverhardware, die Kosten für den Platz im Rechenzentrum, den Strom für Betrieb und Kühlung der Server, die Netzwerkbandbreite und die Gehälter für das Bereitstellungs- und Wartungspersonal. Bei einigen Serviceprovidern können auch variable Kosten für Softwarelizenzen oder Dienste Dritter anfallen. Kunden, die ihre Dienste selbst hosten wollen, tragen diese Kosten direkt.

Durch die Wahl von Open Source Software können Serviceprovider und Kunden die Lizenzkosten im Vergleich zu proprietärer Software senken oder ganz vermeiden, da für Open Source Software definitionsgemäß keine Gebühren für die Softwarelizenz anfallen. Der Einsatz von Open Source Software zur Bereitstellung von Diensten kann auch Entwicklungszeit und -ressourcen einsparen, und die von der Community betriebene Wartung und Aktualisierung kann die langfristigen Kosten senken. Es ist jedoch wichtig, die Gesamtbetriebskosten (TCO) der Open Source Software zu berücksichtigen, einschließlich möglicher Kosten für Support, Wartung und Integration.

Geführte Übungen

1. Nennen Sie einige Möglichkeiten, wie Serviceprovider Einnahmen erzielen können, ohne den Benutzern des Dienstes Gebühren in Rechnung zu stellen.

2. Was könnten Gründe sein, dass ein Serviceprovider seine Software für das Self-Hosting durch Kunden anbietet?

3. Warum sollte ein Serviceprovider seine Software unter der GNU Affero General Public License veröffentlichen?

Offene Übungen

1. Was ist eine Acceptable Use Policy (AUP)?

2. Was ist ein End User License Agreement (EULA)?

Zusammenfassung

Diese Lektion behandelt Geschäftsmodelle von Serviceprovidern, die sich auf Open Source Software stützen, sowie deren verschiedene Einnahmequellen, darunter Abonnement-, Werbeeinnahmen-, Nutzungsdauer- und Provisionsmodelle. Hervorgehoben werden die Implikationen von Open-Source-Softwarelizenzen für Serviceprovider, einschließlich deren Vorteile und Verpflichtungen. Überlegungen zu Sicherheit, Datenschutz und Zuverlässigkeit von Diensten sowie die Bedeutung von Nutzungsbedingungen (ToS), Service Level Agreements (SLA) und Datenverarbeitungsverträgen (DPA) werden ebenfalls erläutert.

Antworten zu den geführten Übungen

1. Nennen Sie einige Möglichkeiten, wie Serviceprovider Einnahmen erzielen können, ohne den Benutzern des Dienstes Gebühren in Rechnung zu stellen.

   Der Dienst kann Werbeanzeigen einblenden, für die die Inserenten zahlen. Der Dienst kann von Anbietern, die auf der Website Produkte oder Dienstleistungen anbieten (beispielsweise Einzelhandel oder Reisen) eine Provision verlangen. Der Serviceprovider kann Kundendaten verkaufen, eine Praxis, die viele Kunden ablehnen würden und die in den Nutzungsbedingungen erläutert sein sollte.

2. Was könnten Gründe sein, dass ein Serviceprovider seine Software für das Self-Hosting durch Kunden anbietet?

   Potenzielle Kunden können die Software ausprobieren, um ihre Funktionen und ihre Zuverlässigkeit zu prüfen, bevor sie sich bei dem Serviceprovider anmelden. Kunden können auch Fehler finden und sogar Korrekturen vorschlagen.

3. Warum sollte ein Serviceprovider seine Software unter der GNU Affero General Public License veröffentlichen?

   Die GNU Affero GPL verlangt von anderen Serviceprovidern, dass sie alle Änderungen, die sie an der Software vornehmen, unter derselben Lizenz veröffentlichen. Diese Bestimmung verhindert, dass Konkurrenten den Serviceprovider ausnutzen, indem sie ein paar Verbesserungen hinzufügen, die sie für sich behalten, und dann ihre Version des Dienstes als besser als das Original anpreisen.

Antworten zu den offenen Übungen

1. Was ist eine Acceptable Use Policy (AUP)?

   Eine Acceptable Use Policy (AUP) ist eine Richtlinie, die die angemessene und unangemessene Nutzung eines Dienstes definiert, um Missbrauch zu verhindern und eine sichere und zuverlässige Arbeitsumgebung zu gewährleisten. Zu den üblichen Punkten einer AUP gehören eine Beschreibung bestimmter Handlungen, die nicht erlaubt sind, wie etwa Spamming oder Hacking, die Verpflichtung der Benutzer, den Dienst verantwortungsvoll zu nutzen, Maßnahmen, die der Serviceprovider bei Verstößen ergreifen kann (wie beispielsweise die Sperrung eines Benutzerkontos), sowie Verfahren zur Meldung und Behandlung von Verstößen.

2. Was ist ein End User License Agreement (EULA)?

   Ein End User License Agreement (EULA), also eine Endnutzer-Lizenzvereinbarung, ist ein Vertrag zwischen einem Anbieter proprietärer Software und einem Endnutzer (nicht einem Unternehmen oder einer Organisation), der dem Benutzer das Recht einräumt, die Software unter bestimmten Bedingungen zu nutzen. Zu den üblichen Punkten von EULAs gehören der Umfang und die Beschränkungen der Lizenz (beispielsweise persönliche Nutzung oder nicht übertragbare Lizenz), verbotene Handlungen (beispielsweise Reverse Engineering oder Weiterverbreitung), Bedingungen, unter denen die Lizenz gekündigt werden kann, sowie das Eigentum und der Schutz der Rechte an geistigem Eigentum. Vollständig open-source-basierende Dienste haben eine Open-Source-Softwarelizenz anstelle einer proprietären EULA.

054.3 Compliance und Risikominderung

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 054.3

Gewichtung

3

Hauptwissensgebiete

- Verständnis, wie die Einhaltung von Lizenzbestimmungen sichergestellt werden kann
- Verständnis, wie man Informationen über Lizenzen pflegt
- Verständnis des Konzepts der Open Source Program Offices
- Verständnis der Auswirkungen von Urheberrecht, Patenten und Warenzeichen auf Open-Source-Geschäftsmodelle
- Bewusstsein für rechtliche Risiken im Zusammenhang mit Open-Source-Geschäftsmodellen
- Bewusstsein für finanzielle Risiken im Zusammenhang mit Open-Source-Geschäftsmodellen

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Software Composition Analysis (SCA)
- Software Bill Of Materials (SBOM)
- Software Package Data Exchange (SPDX)
- OWASP CycloneDX
- Open Source Program Offices (OSPO)
- Produktgarantie
- Produkthaftung
- Ausfuhrbestimmungen

- Auswirkungen von Fusionen und Übernahmen

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	054 Open-Source-Geschäftsmodelle
Lernziel:	054.3 Compliance und Risikominderung
Lektion:	1 von 1

Einführung

Eine bekannte Redensart in der Open-Source-Welt lautet: “Freie Software gibt es nicht umsonst.” Während freie und Open Source Software für Innovation und Kreativität steht, müssen Benutzer und Entwickler eine Reihe von Regeln einhalten. Diese Lektion behandelt die Einhaltung von Regeln, also die Compliance, im Umgang mit Open Source Software sowie Risiken und deren Vermeidung.

Hier geht es um grundlegende Maßnahmen, die Entwickler zur Einhaltung von Lizenzbestimmungen ergreifen müssen, um Risiken beim Einsatz von Open Source Software sowie um Wege, die von einer Organisation verwendete Software zurückzuverfolgen und zu katalogisieren. Wir werden uns ansehen, wie diese Aufgaben in Richtlinien gefasst und von einem Open Source Program Office (OSPO) gefördert werden.

Voraussetzungen für die Freigabe von Software, die auf Open-Source-Komponenten basiert

Wenn eine Organisation Software intern einsetzt, stellen die Lizenzen für freie und Open Source

Software keine Anforderungen. Die Organisation könnte ihre eigenen Regeln festlegen, um Schwachstellen zu vermeiden, um sicherzustellen, dass alle dieselben Komponenten verwenden, oder zu anderen Zwecken. Aber dies ist nicht Thema dieser Lektion. Alle Anforderungen, die die Open-Source-Lizenzen oder die Organisation selbst an die Verwendung der Software stellen, sollten dokumentiert werden, und die Organisation sollte Schulungen zu ihren Richtlinien anbieten.

Selbst wenn die Organisation Software auf ihrem Webserver betreibt und Dienste anbietet, mit denen Personen außerhalb der Organisation interagieren, stellen die meisten Lizenzen für freie und Open Source Software keine besonderen Anforderungen. Die GNU Affero General Public License verlangt jedoch auch für Software, die als Dienst läuft, die Einhaltung des Copyleft-Prinzips.

Die rechtlichen Anforderungen der wichtigsten Lizenzen für freie und Open Source Software wurden bereits in anderen Lektionen behandelt, so dass dieser Abschnitt lediglich eine Liste von Maßnahmen beschreibt, die Entwickler und Manager ergreifen sollten, um diese einzuhalten:

Open-Source-Komponenten prüfen

Ein Unternehmen benötigt klare Richtlinien und Informationen darüber, welche Open Source Software eingesetzt und wo sie in die Produkte integriert werden soll. Solche Richtlinien umfassen die Einhaltung von (Lizenz-)Vorschriften sowie andere Aspekte wie die Gewährleistung der Sicherheit oder das Engagement in der Community, die die Software entwickelt.

Ursprüngliche Lizenz im Code belassen

Entwickler dürfen die Lizenz nicht aus der von ihnen verwendeten Komponente entfernen. Die Aufnahme der Lizenz in den Quellcode ist in der Regel von der Lizenz gefordert. Das Entfernen der Lizenz könnte sogar als Plagiat angesehen werden, da der Entwickler den Anschein erweckt, er habe den Code selbst geschrieben. Das Entfernen der Lizenz könnte auch später zu ernsthaften Problemen führen, da das Unternehmen die Lizenz verletzt, wenn es den Code in seine Produkte aufnimmt.

Überblick über die Lizenzen behalten

Die Organisation muss einen Katalog pflegen, aus dem die Lizenz jeder von der Organisation verwendeten Datei oder jedes Pakets hervorgeht, um sicherzustellen, dass sie die Lizenzen einhält.

Autoren nennen

Fast alle Lizenzen verlangen, dass der Urheberrechtsinhaber (oft als “Autor” bezeichnet) genannt wird, wenn die Software besprochen und beworben wird. Jede Lizenz erklärt, was in diesem Hinweis enthalten sein muss. In der Regel wird ein Standard-Copyrighthinweis mit dem

Jahr und dem Namen des Urheberrechtsinhabers verlangt. Dieser Hinweis sollte in jeder Dokumentation erscheinen, die sich auf das Produkt bezieht, das die Komponente verwendet, einschließlich Werbung und der Website für das Gerät oder Programm.

Keine Billigung implizieren

Einige BSD-Lizenzen verlangen, dass die Person, die ein Produkt mit den Komponenten vertreibt, sicherstellt, dass sie nicht den Eindruck erweckt, der Urheberrechtsinhaber unterstütze das Produkt. Selbst wenn dies nicht explizit gefordert ist, sollte es aus ethischen Gründen befolgt werden.

Quellcode für Software unter Copyleft-Lizenzen bereitstellen

Wenn eine Organisation eine Binärdatei mit Copyleft-Komponenten vertreibt, sei es als Softwaredistribution oder auf einem Gerät, muss die Organisation der Öffentlichkeit den Quellcode dieser Komponenten bereitstellen. Wenn die Organisation den Code ändert und das abgeleitete Werk in Binärform weitervertreibt, ist der Quellcode der veränderten (*abgeleiteten*) Version ebenfalls der Öffentlichkeit zur Verfügung zu stellen. Die Bereitstellung kann auf allen Wegen erfolgen, die der Öffentlichkeit den Zugang erleichtern, etwa durch Veröffentlichung des Codes auf einer Website, einer DVD oder einem USB-Stick.

Keine Copyleft-Komponenten in proprietäre Produkte einbinden

Wenn die Organisation Copyleft-Komponenten in ein Produkt einbindet, muss sie unter bestimmten Umständen das gesamte Produkt unter derselben Copyleft-Lizenz freigeben. Die Bedingungen, die diese Anforderung auslösen, variieren von Lizenz zu Lizenz. Es ist jedoch manchmal nicht eindeutig, welche Art der Verwendung die Copyleft-Anforderung auslöst. Mit Ausnahme von eindeutigen Situationen wie der Verwendung einer Open-Source-Bibliothek (die nicht die Reziprozität des Copyleft auslösen sollte), sollten Entwickler also sehr vorsichtig beim Einsatz von Copyleft-Komponenten sein, es sei denn, die Organisation ist bereit, ihr Produkt unter derselben Lizenz freizugeben.

Code-Änderungen dokumentieren

Wenn Sie veränderte Code-Versionen weitergeben, geben Sie die von der Organisation vorgenommenen Änderungen deutlich an.

Patentrechte gewähren

Einige Lizenzen für freie oder Open Source Software fordern die Gewährung von Patentnutzung für die Software. Wenn also eine Organisation Code unter einer solchen Lizenz freigibt und ein Patent auf einen Prozess im Code besitzt, kann die Organisation von niemandem, der den Code verwendet oder anpasst, eine Gebühr verlangen oder anderweitig Kontrolle auf der Grundlage ihres Patents ausüben.

Markenzeichen respektieren

Einige Open Source Software ist durch Markenzeichen geschützt. Markenzeichen können sich auf Wörter und Phrasen, Logos und andere Bilder und viele andere Dinge beziehen. Einerseits muss eine Organisation das Markenzeichen für jede Software, die sie nutzt, richtig handhaben: Eine häufige Verletzung ist die Parodie, Veränderung oder einfache Darstellung eines Markenzeichens ohne Erlaubnis. Andererseits muss die Organisation, wenn sie das Markenzeichen verwenden möchte, die Anforderungen des Markenzeicheninhabers erfüllen, was Änderungen an der Software ausschließen kann.

Risiken von Open Source Software

In dieser Lektion geht es um die Einhaltung von Vorschriften, daher werden wir uns auf die Risiken in diesem Bereich konzentrieren, aber auch einige andere Themen ansprechen.

Lizenzverstöße

Lizenzen für freie und Open Source Software müssen genauso ernst genommen werden wie andere Lizenzen und Verträge. Organisationen setzen sie durch, wie zum Beispiel die Software Freedom Conservancy, die im Namen kleiner Copyleft-Projekte handelt, die keine eigenen Prozesse zur Durchsetzung haben.

Für diese Organisationen sind gerichtliche Auseinandersetzungen in der Regel das letzte Mittel. Die meisten Verstöße sind unbeabsichtigt und lassen sich durch Aufklärung leicht beheben. Dennoch schadet es dem Ruf einer Organisation, wenn sie in Unkenntnis und unsensibel gegenüber der Community handelt, von deren Software sie abhängig ist.

Tatsächlich wurden schon Klagen eingereicht, wenn ein Benutzer der Software die Zusammenarbeit verweigert und die Software oder der Beklagte namhaft sind.

Selbst wenn ein Unternehmen nicht verklagt wird, kann der Schaden beträchtlich sein, den ein Lizenzverstoß für die Beziehung zur Community und den Ruf des Unternehmens bedeutet. Ein Projekt kann schon daran scheitern, dass Fragen von Entwicklern in den Foren, die der Software gewidmet sind, unbeantwortet bleiben.

Es kann verheerend sein, wenn jemand Copyleft-Software in einem proprietären Produkt findet. Um die Lizenz einzuhalten, müssen die Entwickler entweder den gesamten Copyleft-Code entfernen oder ihr Produkt unter der Copyleft-Lizenz freigeben. Die Freigabe der Software und die Bereitstellung des Quellcodes könnte fatale Auswirkungen auf Geschäftsmodelle haben, die auf Lizenzgebühren oder anderen Arten der Monopolisierung des Produkts basieren.

Vertrauen und Reputation

Wir haben gesehen, dass Lizenzverstöße großen Schaden anrichten können. Auch andere Verhaltensweisen, die das Vertrauen und den Ruf schädigen, bergen Risiken.

Manche Organisationen veröffentlichen Software unter einer freien oder Open-Source-Lizenz und kündigen dann irgendwann an, dass künftige Versionen unter einer proprietären Lizenz stehen werden. Dieser Wechsel kann verlockend sein, da viele Open-Source-Projekte nicht genügend finanzielle Unterstützung von Kunden erhalten.

Solche Lizenzänderungen sorgen jedoch sowohl bei Kunden als auch bei externen Entwicklern, die an dem Projekt mitarbeiten, für Unmut. Manchmal übernehmen externe Entwickler die freie Version und entwickeln sie unabhängig weiter, was als *Forking* bezeichnet wird. Die freie Version konkurriert dann mit der proprietären Version des Unternehmens und könnte Kunden abwerben.

Das Engagement in Projektforen birgt ebenfalls Risiken. Ein Unternehmen muss den dort angemessenen Auftritt lernen. Zu den häufigsten Problemen gehören:

- Der Versuch, finanzielle Beiträge oder den Code des Unternehmens als Druckmittel einzusetzen, um das Projekt in eine Richtung zu drängen, die andere Entwickler nicht wollen.
- Zu hoher Druck auf die Community, beispielsweise durch zu viele Feature Requests oder zu viele Fragen.
- Unhöfliches Verhalten und Verstöße gegen den Verhaltenskodex des Projekts.
- Der Versuch, den Auftritt des Projekts zu dominieren oder auf andere unangemessene Weise Nutzen aus dem Erfolg des Projekts zu ziehen.

Unrentable Investitionen

Geschäftsmodelle werden in einer anderen Lektion behandelt; dieser Abschnitt weist lediglich auf finanzielle Risiken bei der Beteiligung an Open-Source-Projekten hin.

Unternehmen starten oder beteiligen sich an Open-Source-Projekten in der Erwartung, durch Support-Verträge, SaaS-Verträge, Datenerfassung oder sogar Spenden und Zuschüsse Einnahmen zu erzielen. Allerdings sind Open-Source-Projekte häufig weniger lukrativ als erhofft. Natürlich geht jedes Unternehmen mit einem neuen Projekt ein Risiko ein, aber im Open-Source-Umfeld sind zuverlässige Einnahmequellen besonders schwierig zu finden.

Open Source scheint am nachhaltigsten, wenn es andere Ertragsmodelle unterstützt, beipsielsweise den Verkauf von Hardware, Autos, Druckern usw.

Forks

Wie bereits beschrieben, sind die Mitglieder eines Projekts manchmal uneins über den Projektplan, die Leitung oder andere Aspekte und einige gründen ein alternatives Projekt auf derselben Codebasis. Ein solcher *Fork* kann auch von einer Organisation ausgehen, um eine spezialisierte Version der Software zu erstellen. Android nutzt beispielsweise eine spezialisierte Version von Linux, die von Google verwaltet wird. (Google leistet auch viele Beiträge zur Kernversion von Linux, die nicht immer übernommen werden.)

Organisationen können aus verschiedenen Gründen einen Fork erstellen: Die von ihnen an das Kernprojekt übermittelten Änderungen werden abgelehnt, weil andere Entwickler sie nicht wollen; bei einem Projekt unter einer permissiven Lizenz oder einem Projekt, das nur innerhalb der Organisation verwendet wird, sollen Änderungen geheim bleiben.

Das Risiko eines Forks besteht darin, dass die Entwickler des abgespaltenen Projekts nun für die Wartung des gesamten Produkts verantwortlich sind. Wenn zum Kernprojekt wichtige Bugfixes oder neue Funktionen hinzugefügt werden, müssen die Entwickler des Forks diese Änderungen duplizieren oder darauf verzichten. Je mehr Zeit vergeht und je weiter sich die Projekte voneinander entfernen, desto schwieriger wird es, mit den Änderungen des Kernprojekts Schritt zu halten.

Inkompatible Lizenzen

Wie bereits erläutert, enthalten einige Lizenzen für freie und Open Source Software inkompatible Klauseln, so dass solche Komponenten nicht zusammen in einem Produkt verwendet werden können.

Dieses Problem tritt häufig bei Fusionen oder Übernahmen auf: Jedes Unternehmen hat möglicherweise Software mit einer bestimmten Lizenz verwendet, und um den Code in ihren Projekten zu kombinieren, müssen sie sicherstellen, dass die Lizenzen kompatibel sind.

Produkthaftung

Viele Open-Source-Softwarelizenzen (wie auch die Nutzungsbedingungen für viele proprietäre Software und Dienste) schließen ausdrücklich jegliche Haftung für die Nutzung der Software aus, oder anders ausgedrückt: Die Lizenz oder die Nutzungsbedingungen bieten keine Garantie oder Gewährleistung.

Softwarehersteller werden für Probleme mit ihrer Software selten zur Verantwortung gezogen, dennoch klagen Kunden gelegentlich. Gerichte müssen Klauseln zum Ausschluss von Gewährleistung nicht in jedem Fall anerkennen.

Gesetze und Vorschriften erlegen den Entwicklern von Software immer mehr Verantwortung auf. Diese rechtlichen Beschränkungen — oder zumindest Vorschläge für Beschränkungen — sind derzeit vor allem im Bereich der künstlichen Intelligenz (KI) zu beobachten, sind aber manchmal auch weiter gefasst.

Ausfuhrbestimmungen

Viele Länder schränken die Ausfuhr von Waren und Software ein. Bei Software gelten solche Vorschriften vor allem für Sicherheitsprodukte und insbesondere für die Verwendung von Kryptografie. In den Vereinigten Staaten gab es früher strenge Kontrollen für Software, die Kryptografie enthielt, aber die Bestimmungen wurden für Open-Source-Projekte gelockert.

Unternehmen sollten die Ausfuhrbestimmungen des Landes, in dem sie Software entwickeln, kennen; diese Bestimmungen können den Vertrieb ihrer Produkte betreffen.

Software Bill of Materials: Wissen, was man benutzt

Viele Produkte werden mit einer *Bill of Materials*, also einer Art "Stückliste" ausgeliefert, die alle Komponenten auflistet. Wenn Sie beispielsweise ein Regal kaufen, liegt diesem ein Blatt bei, auf dem alle Teile bis hin zu den Schrauben und Muttern aufgeführt sind. Die Liste kann auch nützliche Informationen wie die Abmessungen eines Teils und dessen Produktnummer enthalten, damit Sie es bei Bedarf bestellen können.

Open-Source-Projekte enthalten inzwischen eine *Software Bill of Materials* (SBOM, ausgesprochen "Es-Bom"). Eine SBOM listet zumindest Pakete, Dateinamen, Lizenzen, Autoren oder Eigentümer und Versionsnummern auf. Viele SBOMs gehen noch weiter und enthalten Informationen über Sicherheitslücken.

Projekte generieren mit Hilfe automatisierter Werkzeuge eine SBOM für jedes Release. Benutzer können die SBOM nach den gewünschten Informationen scannen. Die Extraktion von Lizenzen hilft einem Unternehmen beispielsweise, schnell zu entscheiden, ob die Komponenten mit dem eigenen Code kompatibel sind und ob es rechtlich zulässig ist, die Komponenten mit dem eigenen Code zu verwenden.

In ähnlicher Weise hilft das Extrahieren von Versionsinformationen zu jedem Paket einem Unternehmen herauszufinden, ob Teile seines Produkts von verschiedenen Versionen einer bestimmten Bibliothek abhängen. Die Verwendung von zwei Bibliotheksversionen führt zumindest zu einer Aufblähung des Codes, kann aber auch zu Verwirrung und Fehlern führen, wenn eine Komponente mit der falschen Version gebaut wird.

Standards für SBOMs

Da in IT-Umgebungen enorme Mengen (manchmal Zehntausende) von Komponenten verwendet und häufig geändert werden, müssen SBOMs extrem gut strukturiert sein, um Informationen automatisiert und schnell auslesen zu können. Hier geht es um zwei der beliebtesten Formate in der Open-Source-Welt:

Software Package Data Exchange (SPDX)

Das Format stellt jedes Paket und den gesamten Inhalt des Pakets in einer Baumstruktur dar. Das Format dokumentiert Abhängigkeiten zwischen Dateien und andere Beziehungen. Es können Zeiger auf Informationen, sogenannte “Snippets”, erstellt und dann im gesamten Dokument verwendet werden, so dass Informationen nur an einer Stelle definiert werden müssen. Dieses Format wurde von der Linux Foundation entwickelt.

CycloneDX

Dies ist ein größeres Format mit detaillierteren Feldern, insbesondere für Informationen über Sicherheitslücken. Das Format wurde vom *Open Worldwide Application Security Project* (OWASP) entwickelt und ist beim Militär und anderen Organisationen mit hohen Sicherheitsanforderungen beliebt. Ein Eintrag für eine Datei könnte beispielsweise den Namen einer Sicherheitslücke, die Quelle der Informationen über die Sicherheitslücke, betroffene Ziele usw. enthalten. Dieses Format ist auch für Cloud Deployments gedacht, die Tausende verschiedener Systeme umfassen können.

Sowohl SPDX als auch CycloneDX erstellen hierarchische Strukturen in verschiedenen Formaten, darunter JSON und XML. Für jeden Standard gibt es zahlreiche Tools, sowohl zur Erstellung der Formate als auch zum Scannen der erstellten Dateien nach Informationen. Beliebte Websites zur Versionskontrolle ermöglichen es Entwicklern, mit einem Mausklick eine SBOM in einem dieser Formate zu erstellen.

Software Composition Analysis

Um die von einer Organisation eingesetzte Software bewerten zu können, benötigt man Informationen, woher sie stammt, welche Schwachstellen sie enthält, welche Lizenzen sie verwendet und andere Details. Diese Aufgabe wird als *Software Composition Analysis* (SCA) bezeichnet, und es gibt zahlreiche Tools, die anspruchsvolle Scans von SBOMs und der Software selbst durchführen.

Einige Tools extrahieren Lizenzinformationen, mit deren Hilfe ein Unternehmen entscheiden kann, ob die Software sicher in ein Produkt integriert werden kann und ob verschiedene Komponenten kompatible Lizenzen haben. Einige Tools vergleichen sogar Codeschnipsel mit Bibliotheken gängiger Open-Source-Projekte, um herauszufinden, ob Code aus den Open-Source-

Projekten ohne korrekte Namensnennung übernommen wurde.

Der Einsatz dieser Tools ist vor allem bei Fusionen oder Übernahmen von entscheidender Bedeutung: Das übernehmende Unternehmen könnte feststellen, dass die Software, die es erwerben möchte, weniger wertvoll ist als erwartet, weil sie Open-Source-Komponenten enthält, die Anforderungen stellen, die die beabsichtigte Verwendung im neuen Unternehmen beeinträchtigen.

Formale Richtlinien und Compliance

Die hier beschriebenen Prozesse und Techniken sollten von Managern mit Hilfe von Entwicklern, Anwälten und anderen Experten gestaltet werden. Es geht um Grundsatzentscheidungen, die Organisationen hinsichtlich Open Source Software treffen müssen. Abschließend beschreiben wir daher die Rolle eines *Open Source Program Office* (OSPO), das als Unterstützer und Informationsquelle für entsprechende Richtlinien dienen kann.

Grundsätze für Nutzung von und Beiträge zu Open Source Software

Organisationen können von Open Source Software und der Mitwirkung daran profitieren, müssen dabei aber ihre eigenen Interessen schützen und gleichzeitig Open-Source-Projekte unterstützen. Es sollte daher Richtlinien geben, die folgende Aspekte berücksichtigen:

- Einsatzbereiche der Open Source Software (Betrieb, interne Projekte, Produkte für Kunden usw.)
- Vorteile des Open-Source-Projekts für die Organisation (Funktionen, Leistung, Sicherheit, zukünftige Erweiterungen und Dynamik in der Community).
- Scans für die SCA und Ablage der entsprechenden Dokumentation.
- Belohnungen für Entwickler, die zur Open Source Software beitragen, einschließlich ihrer Teilnahme an öffentlichen Foren.
- Leitlinien für die Beteiligung an Projekten, einschließlich der Frage, wie man das Unternehmen in der Öffentlichkeit vertritt.
- Dokumentationsanforderungen, damit Entwickler außerhalb des Kernteams verstehen, wie sie einen Beitrag leisten und interagieren können.

Das OSPO koordiniert die Erstellung dieser Richtlinien und unterstützt bei der Einhaltung der Vorgaben.

Vereinbarungen für Mitwirkende

Open-Source-Projekte müssen sicherstellen, dass die Beiträge rechtmäßig sind, dass zum Beispiel der Code nicht von einem proprietären Produkt oder einem Open-Source-Projekt mit einer inkompatiblen Lizenz übernommen wurde. Viele Open-Source-Projekte verlangen von Entwicklern darum die Zustimmung zu einem *Contributor License Agreement* (CLA), also einer Lizenzvereinbarung, um die Legitimität ihres Beitrags sicherzustellen.

Entwickler in einem Unternehmen, die zu diesen Projekten beitragen, könnten die Anwälte des Unternehmens bitten, die CLA zu prüfen und zu genehmigen. Die Anwälte sollten daher die Bedeutung der Klauseln in CLAs verstehen und darauf vorbereitet sein, sie zu überprüfen.

Viele CLAs standen aufgrund ihrer Komplexität in der Kritik und weil sie Schlupflöcher ließen, die es Projekten erlaubten, beigetragenen Code anders als von den Beitragenden gewünscht zu nutzen.

Viele Projekte verlangen von Entwicklern anstelle eines CLA daher die Unterzeichnung eines kurzen Dokuments, das als *Developer Certificate of Origin* (DCO) bezeichnet wird. Es bescheinigt, dass der Entwickler das Recht hat, den Code beizusteuern. Das DCO wird in der Regel nicht einem Anwalt zur Prüfung vorgelegt.

Einige Projekte bitten die Mitwirkenden auch einfach darum, das Urheberrecht bzw. die Nutzungsrechte an ihrem Code in einem *Copyright Assignment Agreement* (CAA) auf das Projekt zu übertragen. Viele Mitwirkende lehnen diese Praxis jedoch ab, weil sie den Code dann nicht mehr in eigenen Projekten verwenden können und dies dem Projekt die Kontrolle überlässt.

Open Source Program Office (OSPO)

Open-Source-Projekte vereinen soziale, technische, rechtliche und organisatorische Aspekte. Derzeit ist das Bewusstsein für diese Faktoren in vielen Organisationen noch nicht so ausgeprägt, dass alle Manager, Entwickler, Anwälte usw. sie umfassend verstehen. Daher profitieren Organisationen sehr von der Einrichtung eines *Open Source Program Office* (OSPO) als zentrale Stelle für Interessenvertretung, Politikgestaltung, Durchsetzung von Richtlinien und Aufklärung.

Als eine Art Allzweckabteilung können OSPOs viele Aufgaben erfüllen, wie zum Beispiel:

Förderung von Open Source

OSPOs können sowohl die Konzepte hinter der Open-Source-Bewegung als auch Vorschläge für den Einsatz von Open Source in ihrem Unternehmen fördern. Sie können Entwickler anhalten, nach Open-Source-Lösungen für technische Probleme zu suchen und sie ermutigen, entsprechende Software zu übernehmen. Sie können Manager dazu bewegen, Entwicklern Zeit für die Mitarbeit an Open-Source-Projekten zu geben und diese Mitarbeit bei

Gehaltserhöhungen und Beförderungen zu berücksichtigen.

Ausarbeitung von Richtlinien

OSPOs können Manager motivieren, Richtlinien zu erstellen und Repositories für ihre Arbeit einzurichten.

Durchsetzung von Richtlinien

OSPOs können Entwickler daran erinnern, Software und SBOMs zu scannen und die Vorgaben des Unternehmens zur Nutzung von Open Source zu befolgen. OSPOs können die Dokumentation über die genutzte Software und deren Einsatz im Unternehmen verwalten.

Bildung

OSPOs können Entwicklern darlegen, wie sie Open-Source-Projekte bewerten und sich daran beteiligen können; sie können Anwälten die Details von Lizenzen und anderen rechtlichen Betrachtungen erläutern und dem Unternehmen helfen, die organisatorischen und kulturellen Veränderungen zu verstehen, die den Einsatz von Open Source Software erleichtern.

Es gibt zahlreiche Dokumente und Online-Ressourcen zur Einrichtung eines OSPO. Kleine Organisationen können dafür auch Dritte beauftragen, die sich auf diesem Gebiet spezialisiert haben.

Geführte Übungen

1. Warum sollte ein Entwickler nicht einfach Code von einem Open-Source-Projekt übernehmen und in seinen eigenen Code integrieren, ohne die Lizenz zu berücksichtigen?

2. Welche Arten von Dokumenten würde ein Entwickler unterschreiben, wenn er einen Beitrag zu einem Open-Source-Projekt leistet?

3. Sie haben eine Open Source Software gefunden, die eine hervorragende Grundlage für Ihre Shop-Website wäre. Dürfen Sie Ihr Logo mit dem ursprünglichen Logo kombinieren, um Ihre Website auffälliger zu gestalten?

Offene Übungen

1. Welche Überlegungen könnten Sie dazu veranlassen, trotz aller Schwierigkeiten einen Fork eines Open-Source-Projekts zu erstellen?

2. Aus welchen Gründen könnten Sie eine Open-Source-Software ablehnen, obwoh Sie Ihren Bedürfnissen entspricht?

3. Sie möchten ein Copyleft-geschütztes Logging-Tool in Ihrem proprietären Produkt verwenden. Gibt es eine Möglichkeit, sie zu kombinieren, ohne dass Sie Ihr Produkt unter der Copyleft-Lizenz anbieten müssen?

Zusammenfassung

In dieser Lektion ging es um Überlegungen vor dem Einsatz von freier und Open Source Software. Es wurde erklärt, wie man sich an Lizenzen hält, welche rechtlichen, rufschädigenden und finanziellen Risiken bestehen und wie man wichtige Richtlinien festlegt und in der eigenen Organisation durchsetzt.

Antworten zu den geführten Übungen

1. Warum sollte ein Entwickler nicht einfach Code von einem Open-Source-Projekt übernehmen und in seinen eigenen Code integrieren, ohne die Lizenz zu berücksichtigen?

   Dies ist in der Regel ein Verstoß gegen die Open-Source-Lizenz und stellt außerdem ein Plagiat und damit eine Urheberrechtsverletzung dar. Die Organisation kann gezwungen werden, die Lizenz einzuhalten, wobei die Folgen von Rufschädigung bis zur Zerstörung ihres Geschäftsmodells reichen können, wenn Copyleft-Code mit proprietärem Code vermischt wurde.

2. Welche Arten von Dokumenten würde ein Entwickler unterschreiben, wenn er einen Beitrag zu einem Open-Source-Projekt leistet?

   Ein Contributor License Agreement (CLA) ist eine Vereinbarung, die der Open-Source-Organisation die Nutzungsrechte an dem Code einräumt. Ein Certificate of Origin ist ein kürzeres und einfacheres Dokument, das bestätigt, dass der Entwickler das Recht hat, den Code beizusteuern. Ein Copyright Assignment Agreement (CAA) überträgt alle Rechte an die empfangende Organisation.

3. Sie haben eine Open Source Software gefunden, die eine hervorragende Grundlage für Ihre Shop-Website wäre. Dürfen Sie Ihr Logo mit dem ursprünglichen Logo kombinieren, um Ihre Website auffälliger zu gestalten?

   Wenn das Projekt sein Logo als Marke geschützt hat, würde die Veränderung wahrscheinlich gegen das Markenrecht verstoßen. Auch wenn das Logo nicht markenrechtlich geschützt ist, könnten Veränderung als respektlos und irreführend empfunden werden.

Antworten zu den offenen Übungen

1. Welche Überlegungen könnten Sie dazu veranlassen, trotz aller Schwierigkeiten einen Fork eines Open-Source-Projekts zu erstellen?

   Wenn Sie mit dem aktuellen Stand des Codes zufrieden sind, müssen Sie vielleicht nicht mit den Änderungen am Kernprojekt Schritt halten. Möglicherweise möchten Sie ein Produkt entwickeln, das sich im Einsatzzweck wesentlich vom Kernprojekt unterscheiden wird, und sind daher bereit, sich vom Kernprojekt zu trennen. Der Code könnte in Ihrem Geschäftsplan so wertvoll sein, dass Ihr Team bereit ist, die vollständige Verantwortung für seine Entwicklung und Wartung zu übernehmen.

2. Aus welchen Gründen könnten Sie eine Open-Source-Software ablehnen, obwoh Sie Ihren Bedürfnissen entspricht?

   Der Code könnte zu viele Fehler und Sicherheitslücken enthalten, die Entwicklergemeinschaft des Projekts könnte nicht gut funktionieren, die Richtung, in die sich der Code entwickelt, könnte Ihnen nicht gefallen, oder die Lizenz könnte nicht mit anderem Code in Ihrem Produkt kompatibel sein.

3. Sie möchten ein Copyleft-geschütztes Logging-Tool in Ihrem proprietären Produkt verwenden. Gibt es eine Möglichkeit, sie zu kombinieren, ohne dass Sie Ihr Produkt unter der Copyleft-Lizenz anbieten müssen?

   Die Integration des Codes für das Tool in Ihren Code kann je nach Lizenz die Gegenseitigkeitsverpflichtung des Copyleft auslösen. Das Logging-Werkzeug muss von Ihrem Produkt getrennt werden, damit Ihr Produkt nicht als Derivat gilt. Es könnte zum Beispiel sicher sein, das Copyleft-Tool als separaten Prozess laufen zu lassen und mit ihm über Message Passing zu kommunizieren.

Linux Professional Institute

Thema 055: Projektmanagement

055.1 Modelle der Softwareentwicklung

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 055.1

Gewichtung

3

Hauptwissensgebiete

- Verständnis der Bedeutung und der Ziele des Projektmanagements in der Softwareentwicklung
- Grundlegendes Verständnis der Wasserfall-Softwareentwicklung
- Grundlegendes Verständnis der agilen Softwareentwicklung, einschließlich Scrum und Kanban
- Verständnis des Konzepts von DevOps

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Phasen in Wasserfallprojekten (Anforderungsanalyse, Geschäftsanalyse, Softwaredesign, Entwicklung, Testen, Betrieb)
- Rollen in Wasserfallprojekten (Projektmanager, Business-Analysten, Software-Architekten, Entwickler, Tester)
- Organisation von Scrum-Projekten (Sprints und Sprint-Planung, Produkt- und Sprint-Backlog, tägliche Scrums, Sprint-Review und Sprint-Retrospektive)
- Rollen in Scrum-Projekten (Product Owner, Entwickler, Scrum-Master)
- Organisation von Kanban-Projekten (Kanban-Boards)

Linux Professional Institute

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	055 Projektmanagement
Lernziel:	055.1 Modelle der Softwareentwicklung
Lektion:	1 von 1

Einführung

Das Leben ist voller Projekte: die Planung eines Kinoabends, eines Ausflugs oder einer Familienveranstaltung, der Wochenplan der Kinder oder das Training für den Sport — Pläne sind zu machen und Aufgaben zu erledigen. Oft entwerfen wir mehrere Szenarien, mit Plan B, Plan C usw. Das ist in der Welt der Softwareentwicklung nicht anders.

Rollen in der Softwareentwicklung

Da erfolgreiches Projektmanagement eine Vielzahl von Fähigkeiten erfordert, ist es sinnvoll, verschiedene Personen für klar definierte Rollen vorzusehen. In den folgenden Abschnitten werden einige der Rollen beschrieben, die bei Softwareprojekten häufig anzutreffen sind.

Projektmanager

Die Leitung eines Projekts ist eine komplexe Aufgabe. Manche Situationen sehen auf den ersten Blick einfach aus, erfordern aber viel Sorgfalt und Erfahrung. Dies ist die Zuständigkeit des *Projektmanagers*.

Zu den Aufgaben des Projektmanagers gehört es, dafür zu sorgen, dass das Projekt rechtzeitig, im Rahmen des Budgets und in akzeptabler Qualität fertiggestellt wird. Auch wenn er keine einzige Zeile Code schreibt, ist der Projektmanager für die Ergebnisse des Teams verantwortlich. Er muss mit Kunden die Fristen abstimmen und mit Mitarbeitern in anderen Rollen sprechen, um sicherzustellen, dass alles planmäßig vorangeht.

Business Analyst und Requirement Engineer

Es gibt keinen Platz für Mikromanagement — zumindest nicht an einem angenehmen Arbeitsplatz. *Business Analysten* (BA) und *Requirement Engineers* (RE), im Deutschen auch *Anforderungsmanager* genannt, unterstützen den Projektmanager intern oder extern im Projektteam. Sie sind verantwortlich für eine umfassende und klare Dokumentation der Anforderungen. Sie sind zugleich das Bindeglied zwischen Kunden und Entwicklern. Kunden kommunizieren in einfacher Sprache, was BA/RE in eine eindeutige Dokumentation überführen, damit Entwickler ihre Arbeit effektiv erledigen können.

Stellen Sie sich vor, Sie werden gebeten, eine “große Torte” zu einer Party mitzubringen. Was bedeutet “groß”? Zehn Stücke oder zwanzig? Vielleicht ist es aber auch eine Hochzeitstorte mit hundert Stücken. Der BA/RE muss Anforderungen wie Mengen genau spezifizieren, damit die Entwickler wissen, dass eine Anwendung beispielsweise für eine bestimmte Anzahl von Benutzern konzipiert ist.

Schon zu Beginn der Anforderungsdefinition wird deutlich, wie wichtig klare Kommunikation ist. Zwar ist sie für jede Zusammenarbet unerlässlich, aber in einem Open-Source-Projekt ist sie vielleicht noch wichtiger, da Menschen mit sehr unterschiedlichem Hintergrund an den verschiedenen Teilen des Projekts arbeiten: Studierende, Rentner, Berufstätige mit zwei Jobs usw. Dennoch muss der Projektfortschritt auch in dieser Umgebung reibungslos sein — und Kommunikation darf diesen Fortschritt nicht behindern.

Entwickler, Architekten und Tester

Um die Anforderungen umzusetzen, braucht ein Softwareprojekt *Entwickler*, die das Produkt auf der Grundlage der Dokumentationen und Designentscheidungen erstellen. Ihre Arbeit wird vom *Architekten* beurteilt, der in der Regel über die umfangreichsten technischen und fachlichen Kenntnisse aller Projektbeteiligten verfügt. Jeder Entwickler, insbesondere die Senior Developer, ist für seinen eigenen Code verantwortlich, aber umfassende Entscheidungen werden vom Architekten getroffen oder genehmigt.

Manchmal werden auch *Tester* benannt, die für alle Arten von Tests, die Dokumentation der Ergebnisse und die Erstellung von Tickets verantwortlich sind.

Planung und Terminierung

Nach den grundlegenden Rollen in einem Softwareprojekt betrachten wir nun dessen wichtigste Phasen: Planung, Terminierung, Implementierung, Wartung/Test und Übergabe. Einige Phasen unterscheiden sich in verschiedenen Softwareentwicklungsmodellen, aber Planung und Terminierung sind allgemeine Themen, die wir besprechen, bevor wir tiefer in verschiedene Modelle eintauchen.

Die *Planung* erfolgt, nachdem die BAs/REs die Anforderungsliste vorgelegt haben. Die Planung besteht aus einem oder mehreren Meetings, in denen geklärt wird, was das Team erreichen will, wie es das erreichen will und wie lange das dauern würde. Normalerweise erfolgt in diesen Meetings auch eine Risikoanalyse.

Die Planer müssen dann die Arbeit *terminieren* und *Meilensteine* (*Milestones*) definieren. Jeder Milestone zeigt dem Team, dass ein großer und wichtiger Teil abgeschlossen ist. Manche Komponenten oder Funktionen können Monate der Entwicklung erfordern, so dass die Planer Urlaube, Feiertage und — vor allem in der kalten Jahreszeit — krankheitsbedingte Ausfälle einkalkulieren müssen, um einen genauen Liefertermin zu gewährleisten und Hektik vor der Abgabe zu vermeiden. Bei einem soliden Zeitplan fühlen sich die Entwickler entspannter und können qualitativ hochwertige Lösungen pünktlich und ohne Pannen liefern.

Tools

Zwei Werkzeuge, die für alle Entwicklungsmodelle und das Projektmanagement im Allgemeinen von Vorteil sind, sind ein *Versionskontrollsystem* (VCS) und eine Plattform für das *Application Lifecycle Management* (ALM), also die Verwaltung des Lebenszyklus eines Projekts oder Produkts. Die Versionskontrolle wird in einer anderen Lektion ausführlich behandelt — hier also einige Bemerkungen zum ALM.

ALM ist ein umfassendes Framework, das den Lebenszyklus einer Softwareanwendung von der Entwicklung über die Wartung bis hin zur Ausmusterung verwaltet. ALM integriert Menschen, Prozesse und Tools, um Zusammenarbeit und Effizienz zu verbessern und sicherzustellen, dass alle Phasen des Lebenszyklus einer Software koordiniert und auf die Unternehmensziele abgestimmt sind. Dieser Ansatz hilft, Qualität, Leistung und Zuverlässigkeit von Anwendungen während ihrer gesamten Lebensdauer zu erhalten.

Nach einem groben Überblick über Rollen und die Phasen des Lebenszyklus geht es im Folgenden um Modelle der Softwareentwicklung.

Wasserfallmodell

Das Wasserfallmodell ist eine der frühesten und einfachsten Methoden der Softwareentwicklung. Man kann sie sich wie eine Treppe vorstellen, bei der jede Stufe abgeschlossen sein muss, bevor es weitergeht. Das Modell kennt jedoch nur eine Richtung, so dass es sich nur für Projekte mit klaren Anforderungen und minimalen oder keinen zu erwartenden Änderungen eignet.

Zwar kann die Einfachheit dieses Modells ein Vorteil sein, allerdings gerät sie rasch zum Nachteil, wenn Flexibilität und Anpassungsfähigkeit gefragt sind — und heutzutage ändert sich meist alles während der Arbeit an einem Projekt.

Wie in den vorangegangenen Abschnitten erläutert, benötigt ein Projekt zu Beginn der Entwicklung die Anforderungen, ein abgeschlossenes Konzept und einen endgültigen Zeitplan für die Arbeit mit Meilensteinen. Im Wasserfallmodell sind dies die Ergebnisse des Anforderungsmanagements (Requirement Engineering) und der Geschäftsanalyse (Business Analysis).

Anforderungsmanagement

In dieser ersten Phase werden alle Projektanforderungen gesammelt und dokumentiert. Dazu gehören ausführliche Gespräche zwischen dem Projektleiter und den Auftraggebern, um deren Bedürfnisse und Erwartungen zu verstehen. Das Ergebnis ist eine umfassende Requirements Specification, also ein Dokument, das die Anforderungen an das System beschreibt.

Geschäftsanalyse

Business Analysts (BAs) untersuchen die Anforderungen aus der Unternehmensperspektive, um sicherzustellen, dass sie mit den Unternehmenszielen übereinstimmen. BAs führen Machbarkeitsstudien, Risikobewertungen und Kosten-Nutzen-Analysen durch, um Durchführbarkeit und Nutzen des Projekts zu bewerten. Die gewonnenen Erkenntnisse dienen der Präzisierung des Projektumfangs und der Projektziele.

Softwaredesign

Softwaredesign ist der Schritt, in dem der Architekt einen detaillierten Entwurf erstellt, an dem sich die Entwickler orientieren. Mit dieser Spezifikation kann das Team mit der Umsetzung der Anforderungen beginnen, also mit der eigentlichen Entwicklungsphase.

Entwicklung

In der Entwicklungsphase findet die eigentliche Magie statt — der Code wird geschrieben. Die

Entwickler folgen gewissenhaft der Dokumentation und den Designentscheidungen und schreiben ihre Teile. Nachdem die Entwickler ihren Code mit anderen Entwicklern oder dem Architekten überprüft haben, kann diese Phase als abgeschlossen betrachtet werden.

Testing

Auf die Entwicklung folgt eine Testphase, die von den Testern geleitet wird. Es gibt verschiedene Arten von Tests, die in einer anderen Lektion beschrieben werden, beispielsweise Unit Tests, Integrationstests, Systemtests und Abnahmetests.

Ziel dieser Tests ist es, Probleme vor der Veröffentlichung aufzudecken und den Entwicklern die Möglichkeit zu geben, sie rechtzeitig zu beheben — und nicht erst, nachdem das Projekt bereitgestellt und veröffentlicht wurde und die Benutzer durch Fehler verärgert und frustriert werden.

Betrieb

Nachdem die Tests bestanden und Fehler behoben wurden, kann das Projekt freigegeben, also in der Produktionsumgebung bereitgestellt werden. Diese Phase kann die Installation, die Konfiguration und manchmal auch die Schulung der Benutzer umfassen. Ist das Projekt einsatzbereit, befindet es sich in einer Wartungsphase, in der das Team Probleme der Benutzer beheben und gegebenenfalls Aktualisierungen bereitstellen kann.

Beurteilung des Wasserfallmodells

Fassen wir Vor- und Nachteile des Wasserfallmodells kurz zusammen.

Das Wasserfallmodell hat folgende Vorteile:

Klare Struktur

Der lineare Prozess ist einfach zu verwalten, zu verstehen und nachzuverfolgen.

Ausführliche Dokumentation

Eine detaillierte Dokumentation jeder Phase hilft dem Team zu verstehen, was während der Entwicklung und der zukünftigen Wartung benötigt wird.

Vorhersehbarkeit

Klar definierte Meilensteine sorgen für einen vorhersehbaren Zeitplan und ein planbares Budget.

Vereinfachtes Projektmanagement

Dank seines linearen und unkomplizierten Ablaufs ist das Modell einfach zu verwalten.

Das Wasserfallmodell hat folgende Nachteile:

Mangelnde Flexibilität

Die lineare Starrheit des Modells erschwert die Umsetzung von Änderungen nach Abschluss der Anforderungsphase.

Spätes Testing

Während der Entwicklungsphase werden Tests nur unregelmäßig oder gar nicht durchgeführt, was dazu führen kann, dass gravierende Probleme zu spät erkannt werden.

Annahme perfekter Anforderungen

Das Modell setzt voraus, dass alle Anforderungen korrekt und eindeutig sind, was unrealistisch sein kann. Das Modell bietet keinen Raum für Überprüfungen während der Entwicklung, die sicherstellen würden, dass die Anforderungen korrekt sind und von den Entwicklern verstanden werden.

Fehlendes Feedback

Erst in der letzten Phase kann ein Kunde etwas über das Produkt sagen. Wenn also etwas (oder vieles) nicht seinen Erwartungen entspricht, taucht das Problem erst ganz am Schluss auf.

Agile Softwareentwicklung, Scrum und Kanban

Um diese Reihe modernerer Softwareentwicklungsmodelle zu erkunden, geht es zunächst um den Begriff *agil*: Agilität drückt die Fähigkeit aus, schnell zu reagieren und sich zu bewegen, sowohl physisch als auch gedanklich; und dies ist auch das Ziel in der Softwareentwicklung.

Agile Softwareentwicklung ist eine Methodik, die auf iterativer Entwicklung, Flexibilität und Zusammenarbeit aufbaut. Sie konzentriert sich darauf, kleine Verbesserungen zu liefern und dabei stets Feedback einzuholen und während der Entwicklung umzusetzen. Dieser Ansatz ermöglicht schnelle Reaktionen, Anpassungen und Antworten, was Zeit spart und sicherstellt, dass das Projekt die Erwartungen der Benutzer und Kunden erfüllt.

Die agile Bewegung startete 2001 mit einem Online-Manifest für agile Softwareentwicklung, in dem die folgenden Grundsätze benannt sind:

> Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen.

> Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:
>
> - *Individuen und Interaktionen* mehr als Prozesse und Werkzeuge
> - *Funktionierende Software* mehr als umfassende Dokumentation
> - *Zusammenarbeit mit dem Kunden* mehr als Vertragsverhandlung
> - *Reagieren auf Veränderung* mehr als das Befolgen eines Plans
>
> Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.
>
> — Manifest für Agile Softwareentwicklung

Scrum

Scrum ist eines der populärsten Frameworks — vielleicht sogar das populärste — in der agilen Softwareentwicklung. Scrum basiert auf den folgenden Bausteinen:

> Kurz gesagt fordert Scrum, dass ein:e Scrum Master:in ein Umfeld fördert, in dem
>
> 1. ein:e Product Owner:in die Arbeit für ein komplexes Problem in ein Product Backlog einsortiert;
> 2. das Scrum Team aus einer Auswahl dieser Arbeit innerhalb eines Sprints ein wertvolles Increment erzeugt;
> 3. das Scrum Team und dessen Stakeholder:innen die Ergebnisse überprüfen und für den nächsten Sprint anpassen;
> 4. diese Schritte wiederholt werden.
>
> — The 2020 Scrum Guide

Aber wer sind Scrum Master und Product Owner? Was sind Product Backlog und Sprint? Sehen wir uns die Rollen und Prozesse genauer an.

Sprints und Sprint-Planung

Ein *Sprint* ist eine typischerweise zwei- bis vierwöchige Entwicklungsphase, in der zuvor definierte Aufgaben und/oder Features fertiggestellt werden. Einigkeit über die Aufgaben wird in der vorangehenden *Sprint-Planung* erzielt. In diesem Prozess trifft das Team Entscheidungen darüber, welche Aufgaben im kommenden Sprint fertiggestellt werden können.

Diese Entscheidungen beruhen auf den vom Product Owners (PO) bestimmten Prioritäten, der das Projekt steuert und oft auch die Mittel dafür bereitstellt.

Product Backlog und Sprint Backlog

Der PO verwaltet also die Prioritätenliste, die alle gewünschten Arbeiten am Projekt enthält. In Scrum wird diese Liste *Product Backlog* genannt. Jedes *Sprint Backlog* enthält die ausgewählten Aufgaben für den aktuellen Sprint aus dem Product Backlog. Das Sprint Backlog führt die vom Team während der Sprint-Planung ausgewählten Elemente auf.

Daily Scrum oder Stand-up

Das *Daily Scrum* oder *Stand-up* ist ein kurzes, effizientes Treffen, bei dem die Teams ihre Fortschritte besprechen, Probleme und Blocker benennen und ihre Pläne für den Tag mitteilen. Ein Scrum findet in der Regel zu Beginn des Arbeitstages statt.

Inkrement

Ein *Inkrement* ist ein Ziel, das während eines Sprints erreicht wird. Jeder Sprint sollte zu einem oder mehreren Inkrementen führen. Die Korrektheit der durch das Inkrement erreichten Aufgabe oder Features sollte durch Testen überprüft werden.

Sprint Review

Der *Sprint Review* ist ein Meeting zur Überprüfung des Sprint-Ergebnisses. Diese Meetings sind in der Regel mehr als Demos; Ziel ist Feedback zu den Ergebnissen des Scrum-Teams. Um das Produktziel zu erreichen, geben die Beteiligten Kommentare und Vorschläge für weitere Anpassungen ab. Ergebnisse dieser Meetings können Änderungen oder Ergänzungen des Product Backlogs sein.

Sprint-Retrospektive

Ziel der *Sprint-Retrospektive* ist die Verbesserung von Qualität und Effektivität — und das nicht nur für das aktuelle Produkt. Die Retrospektive soll versteckte Spannungen innerhalb des Teams aufdecken, fehlende Informationen, die Prozesse verlangsamen, Abhängigkeiten von außen, die gelockert werden sollten, um das Team zu entlasten, usw. Das Team bespricht, was gut gelaufen ist, welche Probleme es gab und welche Lösungen für diese Probleme gefunden wurden (oder auch nicht).

Das Ergebnis ist eine Liste von Problemen mit möglichen Lösungen, die der zuständigen Person zugewiesen werden.

Scrum Master

All diese Meetings leitet der *Scrum Master*, den jedes Scrum Team braucht. Er ist dafür

verantwortlich, die Scrum-Prozesse zu moderieren und dem Team zu helfen, das Produktziel zu erreichen, auch wenn während der Sprints Probleme auftreten. Der Scrum Master leitet nicht nur die Prozesse, sondern fungiert auch als Coach, um eine effektive Kommunikation innerhalb des Teams sicherzustellen.

Product Owner

Der *Product Owner* ist für die Wertmaximierung des vom Scrum Team geschaffenen Produkts verantwortlich. Diese Rolle kann in verschiedenen Organisationen und Teams unterschiedlich sein, aber selbst wenn er Aufgaben delegiert, bleibt der Product Owner für die Ergebnisse verantwortlich.

Erfolg setzt voraus, dass die Organisation die Entscheidungen des Product Owners, die sich im Product Backlog und im Inkrement des Sprint Reviews widerspiegeln, respektiert. Der Product Owner ist eine einzige Person, die die Bedürfnisse der verschiedenen Beteiligten und Interessen im Product Backlog repräsentiert. Wer das Backlog ändern möchte, muss den Product Owner davon überzeugen. Der Product Owner definiert und priorisiert die Produktvision und das Backlog und sorgt für Transparenz, Sichtbarkeit und Verständlichkeit des Product Backlogs.

Scrum Master und Product Owner arbeiten zusammen, um den Projekterfolg voranzutreiben. Ihre Partnerschaft hilft dem Entwicklungsteam, hochwertige Funktionen effizient und wirkungsvoll zu liefern.

Entwickler

Die Entwickler setzen die Anforderungen entsprechend dem vereinbarten Sprint Backlog um. Sie können unabhängig voneinander arbeiten, aber es gibt verschiedene Gelegenheiten, bei denen sie zusammenarbeiten können, beispielsweise beim Pair Programming oder bei der Überprüfung des Codes eines anderen Entwicklers.

Beurteilung des Scrum-Modells

Scrum hat sich bei Softwareteams durchgesetzt, aber auch hier gibt es Vor- und Nachteile.

Das Scrum-Modell hat folgende Vorteile:

Flexibilität
: Die Prozesse sind flexibel und iterativ, so dass Änderungen auf der Grundlage von Feedback möglich sind.

Einbeziehung der Kunden
: Regelmäßiges Feedback gewährleistet die Anpassung an die Kundenbedürfnisse.

Verbesserte Qualität

Häufige Tests und Reviews verbessern die Produktqualität.

Zusammenarbeit im Team

Das Modell legt Wert auf Teamarbeit und Kommunikation durch tägliche Stand-ups und regelmäßige Treffen.

Das Scrum-Modell hat folgende Nachteile:

Notwendige Disziplin

Scrum erfordert die strikte Einhaltung der Abläufe, was eine Herausforderung sein kann.

Ausufernde Anforderungen

Es besteht die Gefahr, dass immer mehr Kundenwünsche hinzukommen und die Veröffentlichung verzögern, wenn der Product Backlog nicht gut verwaltet wird.

Rollenverteilung

Standardrollen wie Scrum Master und Product Owner können missverstanden oder schlecht umgesetzt werden.

Ressourcenintensiv

Tägliche Sitzungen und häufige Reviews können zeitaufwändig sein.

Kanban

Kanban ist ein weiteres beliebtes agiles Framework, das den Schwerpunkt auf Arbeitsvisualisierung, Flussmanagement und Prozessverbesserungen legt. Einer der großen Unterschiede zwischen Scrum und Kanban ist, dass Kanban keine Iterationen mit fester Länge erfordert. Dadurch wird die kontinuierliche Lieferung flexibler und kann unterschiedliche Prioritäten ausgleichen.

In den folgenden Abschnitten sehen wir uns an, was Teams für ein Kanban-Projekt benötigen.

Kanban Boards

Ein Kanban Board ist ein visuelles Hilfsmittel, das den Arbeitsfluss in verschiedenen Phasen abbildet. Die wichtigsten Spalten sind “To Do”, “In Progress” und “Done”. Weitere Spalten können hinzugefügt werden, aber diese drei müssen auf dem Board vorhanden sein. Die Visualisierung hilft Teams, Engpässe zu erkennen und den Status von Aufgaben schnell zu erfassen.

Work in Progress Limit

Ein *Work in Progress Limit* hilft, Multitasking zu vermeiden, und erhöht die Konzentration. Mit diesem Limit gibt es einen Punkt, an dem keine weiteren Aufgaben mehr in der Spalte "In Progress" hinzugefügt werden, so dass die Entwickler nicht mit Aufgaben überlastet werden und sich auf die Aufgaben konzentrieren, an denen sie gerade arbeiten.

Teammitglieder

Die Teammitglieder sind für die Erledigung der Aufgaben und den reibungslosen Ablauf im Kanban-System verantwortlich. Sie arbeiten zusammen, um die Arbeit zu priorisieren und alle auftretenden Probleme zu erkennen und zu lösen.

Kanban Manager

Der *Kanban Manager* beaufsichtigt den Kanban-Prozess und stellt sicher, dass das Team die Kanban-Prinzipien und -Praktiken befolgt. Er moderiert Besprechungen, überwacht die Arbeitsabläufe und hilft bei der Lösung von Problemen.

Beurteilung des Kanban-Modells

Der Kern dieses Modells ist einfach; betrachten wir die Vor- und Nachteile: Das Kanban-Modell hat folgende Vorteile:

Visueller Arbeitsablauf
: Kanban Boards machen Arbeitsstatus und -fortschritt klar erkennbar.

Flexibilität
: Da es keine festen Iterationen gibt, unterstützt das Modell die kontinuierliche Bereitstellung (Continuous Delivery) mit hoher Anpassungsfähigkeit.

Begrenzung der Aufgaben
: Das Work in Progress Limit trägt dazu bei, dass Teammitglieder nicht überlastet werden, und steigert somit Fokussierung und Effizienz.

Kontinuierliche Verbesserung
: Das Modell fördert die regelmäßige Bewertung und Verbesserung der Prozesse.

Das Kanban-Modell hat folgende Nachteile:

Fehlender Zeitrahmen
: Ohne feste Fristen ist das Zeitmanagement schwierig.

Weniger Struktur

Dem Modell kann es an Struktur fehlen, die manche Teams brauchen, um organisiert zu bleiben.

Notwendige Disziplin

Wirksame Work in Progress Limits und das Board Management müssen sorgfältig überwacht werden.

Mögliche Vereinfachung

Die Aufgabenbeschreibungen könnten komplexe Projekte zu stark vereinfachen, wenn sie nicht mit Bedacht umgesetzt werden.

DevOps

DevOps ist ein Softwareentwicklungsansatz, der die Teams aus Entwicklung (Development) und Betrieb (Operations) integriert, um Zusammenarbeit, Effizienz und Geschwindigkeit der Bereitstellung zu verbessern. Das Hauptziel von DevOps ist die Verkürzung des Softwareentwicklungszyklus und die kontinuierliche Bereitstellung bei hoher Softwarequalität. DevOps legt den Schwerpunkt auf Automatisierung, kontinuierliche Integration und kontinuierliche Bereitstellung (Continuous Integration/Continuous Delivery, CI/CD) sowie auf eine enge Zusammenarbeit zwischen traditionell getrennten Teams.

Die Automatisierung von Aufgaben wie Code-Integration, Testen, Bereitstellung und Infrastrukturmanagement ist für DevOps von entscheidender Bedeutung. Die Automatisierung sich wiederholender Aufgaben reduziert Fehler, beschleunigt Prozesse und ermöglicht es Teams, sich auf eher strategische Aufgaben zu konzentrieren.

Zu den DevOps-Praktiken gehört die Einrichtung umfassender Überwachungs- und Protokollierungssysteme, um Probleme zu erkennen, Trends zu analysieren und die Systemzuverlässigkeit zu verbessern.

Beurteilung des DevOps-Modells

Obwohl sich DevOps für viele moderne Softwareprojekte empfiehlt, sind auch hier Vor- und Nachteile zu berücksichtigen.

Das DevOps-Modell hat folgende Vorteile:

Schnelligkeit und Effizienz

Das Modell beschleunigt die Entwicklungs- und Bereitstellungszyklen durch Automatisierung.

Verbesserte Zusammenarbeit: Das Modell überbrückt die Kluft zwischen Entwicklungs- und Betriebsteams.

Kontinuierliche Bereitstellung: Das Modell stellt sicher, dass die Software immer in einem einsatzbereiten Zustand ist, was zu häufigeren Releases führt.

Hohe Zuverlässigkeit

Automatisierte Tests und Überwachung erhöhen die Zuverlässigkeit und reduzieren Fehler.

Das DevOps-Modell hat folgende Nachteile:

Wandel der Arbeitskultur

Das Modell erfordert einen bedeutenden Wandel in Arbeitsabläufen und die Akzeptanz in der gesamten Organisation.

Komplexität

Die Verwaltung von CI/CD-Pipelines und automatisierter Infrastruktur kann komplex sein.

Sicherheitsrisiken

Die kontinuierliche Bereitstellung kann bei unvorsichtiger Handhabung zu Sicherheitslücken führen.

Viele Tools

Die Vielzahl der eingesetzten Tools und Technologien kann erdrückend sein.

Geführte Übungen

1. Was bedeutet Agilität?

2. Wie lautet die Scrum-Definition gemäß dem Scrum Guide?

3. Warum kann Scrum in Bezug auf Kundenfeedback besser sein als das Wasserfallmodell?

4. Was sind die positiven Aspekte von DevOps?

Offene Übungen

1. Warum ist das Wasserfallmodell in einem sich schnell verändernden Umfeld nicht die beste Lösung?

2. Was kann passieren, wenn die Rolle des Scrum Masters schlecht umgesetzt wird?

Zusammenfassung

Die Bedeutung des Projektmanagements bei der Softwareentwicklung, insbesondere bei Open-Source-Projekten, liegt in seiner Fähigkeit, für Struktur zu sorgen, Kommunikation zu verbessern, Rollen zu definieren, Risiken zu managen und Qualität zu gewährleisten. Diese Elemente sind entscheidend für den erfolgreichen Abschluss von Projekten und für die Förderung einer kooperativen und produktiven Entwicklungsumgebung.

In dieser Lektion ging es um das Wasserfallmodell, agile Methoden und DevOps. Die Kenntnis dieser Modelle ist für das Verständnis des Projektmanagements in der Softwareentwicklung unerlässlich. Es gibt keinen goldenen Weg — jedes Projekt ist anders. Besprochen wurden Rollen, Prozesse sowie die Vor- und Nachteile der verschiedenen Ansätze, was dabei helfen kann, das passende Modell für ein Projekt auszuwählen.

Antworten zu den geführten Übungen

1. Was bedeutet Agilität?

   Agilität ist die Fähigkeit, schnell zu reagieren und sich zu bewegen, sowohl physisch als auch mental.

2. Wie lautet die Scrum-Definition gemäß dem Scrum Guide?
   - Ein Product Owner sortiert die Arbeit für ein komplexes Problem in einem Product Backlog.
   - Das Scrum Team setzt einen Teil der Arbeit im Rahmen eines Sprints in ein Inkrement, einen Zwischenschritt mit höherem Wert, um.
   - Das Scrum Team und seine Beteiligten prüfen die Ergebnisse und passen sie für den nächsten Sprint an.
   - Wiederholung
3. Warum kann Scrum in Bezug auf Kundenfeedback besser sein als das Wasserfallmodell?

   Da Feedback bereits während der Entwicklung gesammelt wird, kann das Endprodukt die Erwartungen der Kunden besser erfüllen.

4. Was sind die positiven Aspekte von DevOps?

   Geschwindigkeit und Effizienz — Verbesserte Zusammenarbeit — Kontinuierliche Bereitstellung — Hohe Zuverlässigkeit.

Antworten zu den offenen Übungen

1. Warum ist das Wasserfallmodell in einem sich schnell verändernden Umfeld nicht die beste Lösung?

   Beim Wasserfallmodell wird Feedback erst am Ende der Entwicklung gesammelt. Daher muss jede Anforderung, die von den Entwicklern missverstanden wurde, ganz am Ende korrigiert werden. Wenn sich die Umgebung sehr schnell ändert, sollte dieses Modell nicht verwendet werden, da während des Entwicklungszyklus kein Feedback vorgesehen ist.

2. Was kann passieren, wenn die Rolle des Scrum Masters schlecht umgesetzt wird?

   Eine schlecht umgesetzte Rolle als Scrum Master kann das Team daran hindern, effizient hochwertige Produkte zu liefern, und damit die eigentlichen Vorteile von Scrum untergraben. Dem Team fehlt möglicherweise eine angemessene Anleitung zu den Scrum-Praktiken und -Prinzipien, was zu einer inkonsistenten oder falschen Anwendung des Frameworks führt.

   Ohne einen geeigneten Scrum Master, der Hindernisse aus dem Weg räumt, können Fortschritte verlangsamt und die Produktivität des Teams verringert werden. Die Kommunikation zwischen Teammitgliedern und anderen Beteiligten kann leiden, was zu Missverständnissen und nicht abgestimmten Erwartungen führt. Das Team kann aufgrund ungelöster Probleme, mangelnder Unterstützung und ineffektiver Moderation von Scrum Events seine Motivation verlieren. Unzulänglichkeiten können durch schlecht geführte Meetings, mangelnden Fokus und ineffiziente Sprint-Planung und -Reviews entstehen.

055.2 Produktmanagement / Release Management

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 055.2

Gewichtung

2

Hauptwissensgebiete

- Verständnis gängiger Release-Typen
- Verständnis der Softwareversionierung und der Gründe für Haupt- und Nebenversionen
- Verständnis des Lebenszyklus eines Softwareprodukts, von der Planung, Entwicklung und Freigabe bis zur Ausmusterung
- Verständnis der Dokumentation für Produktversionen

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogram

- Alpha- und Beta-Versionen
- Release-Kandidaten
- Feature Freeze
- Haupt- und Nebenversionen
- Semantische Versionierung
- Roadmaps und Milestones
- Changelogs
- Long Term Support (LTS)
- End of Life (EOL)
- Abwärtskompatibilität

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	055 Projektmanagement
Lernziel:	055.2 Produktmanagement / Release Management
Lektion:	1 von 1

Einführung

Die meiste Software entwickelt sich im Laufe der Zeit weiter. Das ist ja das Schöne an Software: Sie ist leicht zu ändern und erfordert nur eine Netzwerkübertragung, um für alle Benutzer wieder aktuell zu sein. Entwickler fügen neue Funktionen hinzu, beheben Fehler und Sicherheitslücken, portieren die Software auf neue Hardware und ergänzen Schnittstellen zu beliebten Programmen und Diensten. Ändert sich die Software, werden neue *Versionen* oder *Revisionen* veröffentlicht.

Diese Lektion behandelt die Logistik für Planung und Durchführung von Änderungen an Software, die Art und Weise, wie Teams mit mehreren Versionen umgehen, Konventionen für die Benennung der Versionen und andere organisatorische Aspekte der Entwicklung. Dies alles sind Aufgaben des *Projektmanagements*. Die Logistik, die sich speziell auf Zeitpunkt, Benennung und Kontrolle von Releases bezieht, wird als *Release Management* bezeichnet.

Merkmale von Releases

Releases unterscheiden sich in Bezug auf Stabilität, Abwärtskompatibilität und Unterstützung. Wir werden jedes dieser Konzepte in den folgenden Abschnitten untersuchen.

Stable und unstable Releases

Stabilität ist eine wichtige Eigenschaft, die Benutzer von Software erwarten. Die erste Frage, die sich viele stellen, wenn sie von einer neuen Version hören, lautet entsprechend: “Wie stabil ist sie?” Mit anderen Worten: Funktioniert sie zuverlässig, oder stürzt sie ab? Beschädigt sie Daten oder liefert sie falsche Ergebnisse?

Stabilität kann auf einem Spektrum gemessen werden, und viele setzen Software auch gerne ein, wenn sie noch einige Fehler enthält. Aber Entwicklungsteams neigen dazu, die Dinge einfach zu halten, und unterscheiden binär *stable* und *unstable* Versionen.

Stabilität bezieht sich sowohl auf die Fehleranfälligkeit der Software als auch auf sichtbare Veränderungen für Außenstehende bzw. Benutzer. Entwickler könnten eine Version einer Softwarebibliothek als unstable betrachten, weil Funktionen geändert wurden, die Programmierer aufrufen (mit der Folge, dass die Entwickler ihre Software für die nächste Version möglicherweise umschreiben müssen). Die Software könnte aber auch aus Benutzersicht instabil sein, weil Entwickler eine Funktion entfernt haben.

Gibt es Gründe, eine unstable Version zu veröffentlichen? Ja. Sie ist wertvoll, weil Benutzer neue Funktionen schon in einer frühen Projektphase ausprobieren und auf Fehler testen können.

Die meisten Projekte geben sehr frühe Versionen der Software für wichtige Kunden zum Testen frei, die sogenannten *Alpha Releases*. Natürlich sollte man diese Releases nicht in produktiven Arbeitsumgebungen einsetzen — sie sind nur zum Testen gedacht. Tester sollten Alpha Releases auf Rechnern außerhalb der Arbeitsumgebung laufen lassen, da Fehler in diesen Releases die auf dem Computer gespeicherten Daten beschädigen könnten.

Steht die Software nach Ansicht der Entwickler kurz davor, als stabil zu gelten, veröffentlicht das Projekt normalerweise eine weitere Testversion, die als *Beta Release* bezeichnet wird. Diese sollte immer noch nicht auf Produktivsystemen eingesetzt werden, sondern ausschließlich für Tests.

Sowohl bei Alpha- als auch bei Beta-Versionen nutzen Entwickler standardisierte Verfahren, um Fehler zu melden und den Fortschritt bei der Fehlerbehebung zu verfolgen.

Bei manchen Projekten gibt es zwischen der Beta- und der stable Version eine weitere Phase, in der die Entwickler alle Fehler behoben haben und das Produkt als fertig erachten. Sie bezeichnen die Version als *Release Candidate* und stellen sie ausgewählten Kunden oder Interessenten zur Verfügung, damit diese den Einsatz neuer Funktionen planen können.

Es ist wichtig, dass Benutzer neue Funktionen ausprobieren und Feedback dazu geben, bevor die Entwickler die Schnittstellen fertigstellen und neue Funktionen aufwendig stabilisieren.

Abwärtskompatibilität

Die meisten Projekte bemühen sich um *Abwärtskompatibilität* (*Backward Compatibility*): sie versuchen, Funktionen, die in älteren Versionen vorhanden waren, zu erhalten. Kompatibilität kann auf mehreren Ebenen bestehen: Indem Entwickler beispielsweise Abwärtskompatibilität für die Programmierschnittstelle von Anwendungen (API) erhalten, stellen sie sicher, dass alter Quellcode weiterhin funktionieren kann, auch wenn er möglicherweise neu kompiliert werden muss. Wenn Hardwarehersteller oder Betriebssystementwickler Abwärtskompatibilität für das *Application Binary Interface* (ABI) erhalten, sorgen sie dafür, dass alte ausführbare Dateien auf neuen Versionen der Hardware laufen.

Wir werden uns später ansehen, wie Projekte mit fehlender Abwärtskompatibilität umgehen, wenn sie sich also entscheiden, alte Schnittstellen für die neuen Funktionen oder Umgebungen nicht mehr unterstützen.

Support und End of Life (EOL)

Idealerweise wird Software immer besser, wenn Entwickler Probleme entdecken und beheben. Aber mit der Zeit können Funktionen aufgrund von Änderungen in der Softwareumgebung nicht mehr funktionieren. Außerdem werden neue Fehler entdeckt, die zuvor nicht aufgefallen waren.

Neue Versionen kombinieren oft Sicherheits- und Fehlerkorrekturen mit neuen Funktionen. Vielleicht möchten Sie die neuen Funktionen gar nicht haben — vielleicht bevorzugen Sie sogar eine alte Funktion, die die Entwickler entfernt haben, um Platz für neue Funktionen zu schaffen. Aber die meisten Benutzer aktualisieren auf die neue Version für die Sicherheitskorrekturen, die sie vor Angriffen schützen.

Manche nutzen alte Softwareversionen weiter und verzichten auf ein Upgrade, weil die neue Version in ihrer Umgebung anders oder gar nicht funktioniert. Wenn Unternehmen Geld für Upgrades verlangen, ist das für manche Kunden ebenfalls ein Grund, kein Upgrade durchzuführen; bei Open Source muss man für Upgrades allerdings selten etwas zahlen.

Entwickler kommen den Benutzern alter Versionen nach Möglichkeit entgegen, indem sie Fehler in den alten Versionen beheben, ohne die neuen Funktionen hinzuzufügen, die die Software möglicherweise instabil machen. Natürlich können Entwickler dies nicht unbegrenzt leisten; irgendwann werden sie eine alte Version nicht mehr reparieren und die Benutzer auffordern, entweder ein Upgrade durchzuführen oder die Korrekturen selbst vorzunehmen.

Das Beheben von Fehlern und Sicherheitslücken wird als *Support* (Unterstützung) für die Software bezeichnet. Er unterscheidet sich von dem Support, den Helpdesks und andere anbieten, um Benutzer beim Einsatz der Software zu unterstützen. Was das Versionsmanagement angeht, so ist eine *unterstützte Version* eine, für die die Entwickler Fehlerkorrekturen zusagen, während

sie dies bei einer *nicht unterstützten Version* nicht (mehr) tun.

Beachten Sie, dass “Support” meist für Software angeboten wird, die von Unternehmen oder großen Organisationen entwickelt wird. Kleinere Open-Source-Projekte, die stark vom Einsatz Freiwilliger abhängig sind, tun oft ihr bestes, um Fehler zu beheben und neue Versionen herauszubringen, ohne dies garantieren zu können. Sie sehen oft auch keine Notwendigkeit, alte Versionen zu reparieren; da der Code Open Source ist, können Benutzer, die nicht aktualisieren wollen, jemanden dafür bezahlen, eine alte Version zu reparieren.

Wird Support angeboten, veröffentlichen die Entwickler einen Zeitplan, der angibt, wie lange sie eine Version unterstützen werden. Das Datum, zu dem der Support endet, wird als *End of Life* (EOL) der Software bezeichnet. Benutzer können die alte Version bis zum EOL einsetzen und erwarten, dass Fehler behoben werden; nach dem EOL müssen sie jedoch ein Upgrade durchführen oder das Risiko tragen.

Große Projekte, wie beispielsweise die Debian-Distribution von GNU/Linux, bieten Versionen mit *Long Term Support* (LTS) an. Das bedeutet, dass die Entwickler eine bestimmte Anzahl von Jahren Fehler beheben. Kunden, die großen Wert auf Stabilität legen, scheuen sich vielleicht, Softwareversionen zu installieren, die zahlreiche Änderungen enthalten; die Änderungen könnten Prozesse unterbrechen, auf die sich die Kunden verlassen. Solche Kunden, vor allem große Unternehmen und Institutionen, schätzen die Sicherheit von LTS-Versionen.

Softwareversionierung: Major, Minor und Patches

Wir haben gesehen, dass Versionsverwaltung komplex ist: Einige Versionen beheben Fehler, andere fügen neue Funktionen hinzu, und die Versionen können sich in Hinblick auf Stabilität unterscheiden. Entwickler geben durch die Bezeichnung an, wie groß die Änderungen an der Software in einer Version sind. Diese Konventionen, denen fast alle Projekte bei der Kennzeichnung von Versionen folgen, nennt man *semantische Versionierung*.

Nehmen wir die frühe Geschichte des Linux-Kernels als Beispiel: Linus Torvalds bezeichnete die erste stabile Version als 1.0. Als er den Kernel verbesserte, nannte er die nachfolgenden Versionen 1.1, 1.2 usw. Die 1 vor dem Punkt steht für die *Major Version* (Hauptversion), die Zahlen nach dem Punkt bezeichnen die *Minor Version* (Unterversion). Man kann davon ausgehen, dass Version 1.2 mehr Funktionen bietet als Version 1.1.

Aber es gab auch unzählige kleine Veröffentlichungen, die manchmal nur ein paar Fehler behoben. Um anzuzeigen, dass diese Änderungen nur geringe Auswirkungen auf die Funktionalität des Kernels hatten, fügte Torvalds eine dritte Nummer hinzu, die *Patch* (also “Flicken”) genannt wurde. So wurde 1.0 auf 1.0.1 aktualisiert, dann 1.0.2 und so weiter.

Die Patch-Nummerierung beginnt bei einer neuen Minor Version wieder bei 0, und die Nummerierung der Minor Versions startet wieder bei 0, sobald eine neue Major Version erscheint.

Entwickler fassen mehrere Versionen mit einem `x` zusammen; so steht 1.x beispielsweise für alle Versionen unter der Major Version 1.

Aber was ist der Unterschied zwischen einer Änderung, die zu einer neuen Minor Version führt, und einer Änderung, die eine neue Major Version verdient? In der Regel vergehen Jahre, bevor eine neue Major Version veröffentlicht wird, und sie muss bedeutende Aktualisierungen enthalten.

Im Allgemeinen versuchen Entwickler, Abwärtskompatibilität bei Versionsänderungen zu bewahren. Ist dies nicht möglich, sollten sie die Major Version erhöhen.

Bei einer Versionsnummer kleiner als Null — also beispielsweise 0.9 — zeigt die führende Null an, dass es sich um eine frühe Version der Software handelt, die instabil und noch nicht für den produktiven Einsatz geeignet ist. Es gibt auch keine Garantie für Abwärtskompatibilität, bis Entwickler Versionen veröffentlichen, die mit 1 beginnen.

Alpha-Versionen werden in der Regel durch das Anhängen von “a" oder "`alpha” an die Versionsnummer gekennzeichnet, wie etwa 3.6alpha; bei Beta-Versionen ist es entsprechend “b” oder “beta”.

Der Lebenszyklus von Softwareprodukten

Das Leben eines Entwicklers ist keineswegs einfach. Jeder will etwas: Der eine will den Fehler im Bildschirmlayout so schnell wie möglich behoben haben, ein anderer hingegen den Fehler bei der Benennung von Dateien. Benutzer fordern neue Funktionen, und wenn verschiedene Entwickler parallel an verschiedenen Funktionen arbeiten, stellen sie fest, dass die Änderungen des einen die des anderen zunichte machen können.

Projektmanagement und das so genannte Versionsmanagement als ein Teilbereich befassen sich mit diesen Fragen. Normalerweise übernimmt ein leitender Projektmitarbeiter die Verantwortung für diese Managementaufgabe.

Wie Menschen durchlaufen auch Softwareversionen einen Lebenszyklus. Jede Version beginnt mit einer Diskussion über neue Funktionen und andere erforderliche Änderungen, durchläuft Entwicklungs- und Testphasen und wird planvoll ausgeliefert.

Planung und Roadmaps

Entwickler versuchen, weit im Voraus festzulegen, welche Änderungen sie an einem Produkt vornehmen wollen. Im kommerziellen Umfeld sprechen sie mit ihren Kollegen vom Marketing, die filtern und zusammenfassen, was ihre Kunden wünschen. Open-Source-Projekte hängen eher von den Ideen ab, die von Entwicklern und Benutzern eingereicht und in einer Datenbank, dem *Issue Tracker*, erfasst werden. Wenn Benutzer die Behebung eines Fehlers oder eine neue Funktion wünschen, erstellen sie ein sogenanntes *Issue*. Die Entwickler setzen dann Prioritäten für die Änderungen und entscheiden, wer sich um welches Issue kümmert.

Wie aber werden Änderungen ausgewählt und priorisiert? Das kann ein ungeordneter Prozess sein, aber gute Projektmanager in Open-Source-Projekten ermutigen zu breitem Input und stellen gleichzeitig sicher, dass Entscheidungen getroffen werden. Einige Projekte halten sogar Konferenzen ab, auf denen die Teilnehmer die Prioritäten diskutieren.

Eine öffentliche *Roadmap*, also ein Fahrplan für die geplanten Änderungen, legt die Schritte fest. Sie kann viele Versionen und Jahre in die Zukunft abbilden. Jeder Schritt wird als *Milestone* (Meilenstein) bezeichnet und kann mit einem Zieldatum verbunden sein.

Release-Planung

Es gibt grundsätzlich zwei Ansätze, Releases zu planen: zeitbasiert oder funktionsbasiert. Ein Projekt kann ein Release in regelmäßigen Abständen — zum Beispiel alle sechs Monate — planen und alles aufnehmen, was zu diesem Zeitpunkt fertig ist. Alternativ kann das Projekt bestimmte Funktionen für ein Release vorsehen, und den Entwicklern so viel Zeit lassen, wie sie für die Fertigstellung dieser Funktionen benötigen.

Wenn der Zeitpunkt der Veröffentlichung näher rückt, legt der Release Manager die Termine für die Alpha-, die Beta- und die stabile Version fest. Die Teammitglieder prüfen regelmäßig die Fehlerberichte und versuchen, ihre Arbeit so zu planen, dass sie die Meilensteine einhalten.

Um eine neue Version fertigzustellen, muss ein Projekt ab einem gewissen Punkt aufhören, Ideen für neue Funktionen anzunehmen, und sich darauf konzentrieren, die vorhandenen Funktionen korrekt umzusetzen. Dieser Zeitpunkt wird als *Feature Freeze* bezeichnet.

Dokumentation für Produktversionen

Roadmaps erläutern, wie bereits erwähnt, die Funktionen, die die Entwickler in jede Version aufnehmen wollen. Die Version wird von einer Liste der Änderungen begleitet, die als *Changelog* (Änderungsprotokoll) bezeichnet wird. Im Allgemeinen listet das Changelog neue Funktionen, Änderungen an bestehenden Funktionen, Funktionen, die entfernt wurden, Funktionen, die die

Entwickler in Zukunft entfernen wollen (sogenannte *deprecated* Features), und Fehlerbehebungen auf.

Daher können Changelogs recht detailliert ausfallen. Die Benutzer müssen besonders auf die Funktionen achten, die entfernt wurden oder veraltet sind, da sich möglicherweise Programme und die Art der Bedienung ändern. Natürlich versuchen die Entwickler, nur die Funktionen zu entfernen, die niemand mehr benötigt.

Diese Aufgabe ist zeitaufwändig, und es kann leicht passieren, dass Entwickler eine Änderung übersehen und in der Dokumentation vergessen.

Geführte Übungen

1. Was unterscheidet eine stable von einer unstable Version?

2. Würden Sie eine Funktionsänderung zwischen einer Version 2.6.14 und einer Version 2.6.15 erwarten?

3. Würden Sie eine Funktionsänderung zwischen einer Version 2.6.0beta und einer Version 2.6.0 erwarten?

4. Warum sollten Sie nicht davon ausgehen, dass die Version 1.0 abwärtskompatibel mit der Version 0.9 ist?

5. Wenn Sie eine Sicherheitslücke in einer Version nach dem Feature Freeze, aber vor der Veröffentlichung entdecken, können Sie diese beheben lassen?

Offene Übungen

1. Angenommen Sie möchten eine Version einer Open Source Software nach dem EOL weiter verwenden, weil sie eine Funktion enthält, die Sie benötigen. Was können Sie tun, um sie weiterhin zu nutzen?

2. Welche Kriterien führen dazu, dass eine Fehlerbehebung oder ein Feature Request gegenüber anderen bevorzugt wird?

Zusammenfassung

In dieser Lektion wurden die wichtigsten Merkmale von Versionen beschrieben: Stabilität, Abwärtskompatibilität und Unterstützung. Wir haben die Bedeutung von Versionsnamen und -nummern sowie die wichtigsten Aspekte des Versionsmanagements, einschließlich der Dokumentation, besprochen.

Antworten zu den geführten Übungen

1. Was unterscheidet eine stable von einer unstable Version?

   Releases werden in zweierlei Hinsicht als stabil angesehen: Sie funktionieren gut, ohne abzustürzen oder falsche Ergebnisse zu liefern, und die Schnittstelle, die sich Benutzern oder Programmierern präsentiert, ist abwärtskompatibel mit früheren Versionen.

2. Würden Sie eine Funktionsänderung zwischen einer Version 2.6.14 und einer Version 2.6.15 erwarten?

   Nein. Die dritte Zahl in der semantischen Versionierung steht für einen Patch, der einen Fehler behebt oder eine andere kleinere Aufgabe wie eine Neuformatierung erfüllt. Eine Funktionsänderung sollte zu einer neuen Minor oder Major Release führen.

3. Würden Sie eine Funktionsänderung zwischen einer Version 2.6.0beta und einer Version 2.6.0 erwarten?

   Nein. Die Beta-Version ist eine Testversion, die alle Funktionen enthält, die in der endgültigen Version enthalten sein werden.

4. Warum sollten Sie nicht davon ausgehen, dass die Version 1.0 abwärtskompatibel mit der Version 0.9 ist?

   Die Nummer 0.9 weist potenzielle Benutzer ausdrücklich darauf hin, dass sich die Software noch in der Entwicklung befindet und sehr wahrscheinlich beispielsweise ein anderes Interface haben wird, wenn sie als Version 1.0 stabil wird.

5. Wenn Sie eine Sicherheitslücke in einer Version nach dem Feature Freeze, aber vor der Veröffentlichung entdecken, können Sie diese beheben lassen?

   Auf jeden Fall. Der Zeitraum zwischen dem Feature Freeze und der Veröffentlichung ist dazu gedacht, Fehler zu entdecken und zu beheben, einschließlich Sicherheitslücken.

Antworten zu den offenen Übungen

1. Angenommen Sie möchten eine Version einer Open Source Software nach dem EOL weiter verwenden, weil sie eine Funktion enthält, die Sie benötigen. Was können Sie tun, um sie weiterhin zu nutzen?

   Nach dem EOL sind die Projektentwickler nicht verpflichtet, Fehler zu beheben, einschließlich Sicherheitslücken. Daher sollten Sie den Bug Tracker und die Mailinglisten des Projekts aufmerksam verfolgen, um herauszufinden, welche Fehler auftauchen. Da der Code verfügbar ist, können und sollten Sie Fehler, die in Ihrer Version enthalten sind, beheben. Theoretisch könnten Sie sogar neue Funktionen in Ihre Version einbauen. Das würde Ihre Version zu einem Fork des Originals machen.

2. Welche Kriterien führen dazu, dass eine Fehlerbehebung oder ein Feature Request gegenüber anderen bevorzugt wird?

   Bei Fehlerbehebungen gehören zu den Kriterien der Schweregrad (der dem Fehler zugewiesen wird, nachdem er in die Fehlerdatenbank aufgenommen wurde) und die Zahl der davon betroffenen Benutzer. Bei einer Funktion gehören zu den Kriterien die Zahl der Benutzer, die diese Funktion wünschen, der Aufwand, sie zu programmieren, und ihre möglichen Auswirkungen auf andere Teile des Programms.

055.3 Community Management

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 055.3

Gewichtung

2

Hauptwissensgebiete

- Verständnis der Rollen in Open-Source-Projekten
- Verständnis der allgemeinen Aufgaben in Open-Source-Projekten
- Verständnis der verschiedenen Arten von Beiträgen zu Open-Source-Projekten
- Verständnis der verschiedenen Arten von Mitwirkenden an Open-Source-Projekten
- Verständnis der Rolle von Organisationen bei der Pflege von Open-Source-Projekten
- Verständnis der Übertragung von Rechten von Einzelpersonen auf eine Organisation, die ein Projekt betreut
- Verständnis der Regeln und Richtlinien in Open-Source-Projekten
- Verständnis der Zuordnung und Transparenz von Beiträgen
- Verständnis der Aspekte Vielfalt, Gleichberechtigung, Inklusivität und Nichtdiskriminierung

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Software-Entwicklung
- Dokumentation
- Design und Artwork
- Benutzerunterstützung
- Entwickler

- Release-Manager
- Benutzer
- Projektleiter und wohlwollende Diktatoren
- Privatpersonen und Unternehmen
- Enthusiasten und Fachleute
- Mitglieder des Kernteams und gelegentliche Mitwirkende
- Beiträge zu Code und Dokumentation
- Fehlerberichte
- Forks
- Stiftungen und Sponsoren
- Beitragsvereinbarungen
- Developer Certificates of Origin (DCO)
- Kodierungsrichtlinien
- Verhaltenskodizes

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	055 Projektmanagement
Lernziel:	055.3 Community Management
Lektion:	1 von 1

Einführung

Ein wichtiger Grund für den Start eines Open-Source-Projekts sind die Beiträge vieler verschiedener Personen. Mit Open Source ist es einfacher, auf die unterschiedlichen Bedürfnisse und Interessen der am Projekt Beteiligten einzugehen und von deren vielfältigen Fähigkeiten zu profitieren. Daher ist eine diverse Community von zentraler Bedeutung für den Erfolg eines Projekts. In der Community kommen die kreativen Kräfte zusammen, die den Erfolg des Projekts ausmachen.

Diese Lektion beschreibt die Grundlagen von FOSS-Communities und wie Menschen in diesen Gemeinschaften zusammenarbeiten. Da sie aus Individuen bestehen und Projekte unterschiedliche Ziele und Anforderungen haben, ist jede Community natürlich einzigartig.

Rollen in Open-Source-Projekten

Kernziel der meisten Open-Source-Projekte ist Software, so dass für solche Vorhaben zumindest Programmierer, Softwaredesigner oder -architekten, Tester, Release-Manager und andere Experten für die Code-Entwicklung notwendig sind. Auch Dokumentation ist in der Regel ein Teil solcher Projekte, so dass die Community auch Autoren, Redakteure und Korrekturleser benötigt.

Andere Projekte produzieren vielleicht keinen Code, sondern andere “Ergebnisse” beispielsweise ein Buch-, Kunst- oder Musikprojekt oder einen politischen Bericht. Wikipedia ist ein bekanntes Beispiel für die Zusammenarbeit an einem Open-Source-Projekt, das sich auf Textdokumente und Medien konzentriert. Auch diese Projekte benötigen Experten für die Produktion, Gestaltung und Bereitstellung der Ergebnisse.

Communities profitieren von vielen weiteren Fähigkeiten, wie beispielsweise Marketing, Verwaltung, Rechtsberatung, künstlerisches Talent für Logos oder Diagramme in der Dokumentation, aber auch Kenntnisse bei der Pflege der Projektwebseite und anderer Kommunikationsmittel. Open-Source-Projekte können auch persönliche Treffen und sogar Konferenzen organisieren; in diesem Fall brauchen sie Organisatoren, die diese Veranstaltungen mit Leben füllen.

Die folgenden Abschnitte vermitteln einen Eindruck von den Fähigkeiten, die eine Community sucht. Diese allgemeinen Aufgaben lassen sich weiter unterteilen und spezifizieren. Ein einfaches Beispiel ist die Aufgabe eines Autors, der Texte anderer Autoren bearbeitet.

Ein aktives Projekt braucht unter den Programmierern immer einige “Veteranen” (oder *Senior* Programmierer), die den Code und die Programmierstandards gut kennen und mit den *Newbies* (den Neulingen im Projekt) zusammenarbeiten, die neue Ideen einbringen und einfache Aufgaben, wie etwa Fehlerkorrekturen, erledigen. Im besten Fall entwickeln sich einige der Neulinge irgendwann selbst zu Veteranen.

Die Qualität des Codes erfordert sorgfältige Kontrolle darüber, was in das zentrale Repository gelangt, das mit den Benutzern geteilt wird. Daher erhalten einige Veteranen den Status eines *Committers* (“Beiträger”). Sie sind dafür verantwortlich, die Beiträge auf Qualität, Nützlichkeit und Konformität mit den Coding Standards zu prüfen. Sie haben das Privileg, den vorgeschlagenen Code in das vertrauenswürdige Repository aufzunehmen. Andere Mitwirkende reichen den Code bei den Committern zur Überprüfung ein.

Viele Committer und andere Veteranen sind auch *Mentoren* für neue Mitwirkende, um ihnen die Standards und Praktiken zu vermitteln, die sie brauchen, damit ihr Code akzeptiert wird.

Manche Committer wissen das Geschenk nicht zu schätzen, das ein Mitwirkender mit seinem Beitrag macht. Wenn der Beitrag nicht den Qualitäts- oder Kodierungsstandards entspricht, kann der Committer ihn einfach ablehnen. Aber eine Ablehnung kann den potenziellen Mitwirkenden entmutigen und ihm eine wertvolle Möglichkeit nehmen, mit den Standards vertraut zu werden. Gute Committer sind darum auch Mentoren. Sie geben Hinweise, wie der Code den Standards gerecht wird, und arbeiten mit neuen Mitwirkenden zusammen. Ist der Beitrag wirklich unbrauchbar, sollte dem Mitwirkenden zumindest gedankt und gezeigt werden, wie er gegebenenfalls an anderer Stelle zum Projekt beitragen kann. Es ist wichtig, Leuten zu helfen, ihre

Fähigkeiten zu entdecken und zu verbessern.

Wenn ein Unternehmen ein Open-Source-Projekt startet, benennt es manchmal Committer aus den eigenen Reihen, um sicherzustellen, dass es Richtung und Qualität des Projekts kontrollieren kann. Aber oft erlauben Unternehmen auch Externen, Fähigkeiten und Engagement zu zeigen und Committer zu werden.

Projektleiter (*Project Leads*) treffen kritische Entscheidungen, beispielsweise welche neuen Funktionen unterstützt werden sollen. Projektleiter müssen oft auch Entscheidungen treffen, die nicht mit der Programmierung zu tun haben: zur Bewerbung des Projekts, zur Beilegung größerer Meinungsverschiedenheiten und zur Beschaffung von Finanzmitteln. Projektleiter und Committer arbeiten eng mit den *Release Managern* zusammen, um zu entscheiden, wann eine Codebasis stabil ist und mit den Benutzern geteilt werden kann.

Manche Projekte haben einen einzigen Projektleiter, der oft als “benevolent dictator” (“wohlmeinender Diktator”) bezeichnet wird. Dabei handelt es sich in der Regel um eine Person, die das Projekt ins Leben gerufen hat (Linus Torvalds ist ein bekanntes Beispiel für Linux) und die über Autorität oder sogar Charisma verfügt. Die meisten Projekte ziehen es jedoch vor, ein kleines Komitee von Projektleitern einzurichten; sogar das Linux-Projekt ist zu einem Komiteeansatz übergegangen.

Projekte können auch einzelne Mitwirkende bitten, Support in den Projektforen zu leisten. Aber gesunde Projekte sollten viele sachkundige Benutzer haben, die sich gegenseitig unterstützen.

Wir schließen diesen Abschnitt mit einer sehr wichtigen Rolle ab — der des *Community Managers*. Er sorgt für offene und konstruktive Kommunikation und stellt sicher, dass die Community ihr Ziel im Auge behält. Der Community Manager weiß, wie man Mitwirkende gewinnt und motiviert, Streitigkeiten löst, Mitglieder der Community vor Anfeindungen schützt und das Wohl der Gemeinschaft fördert.

Aufgaben in Open-Source-Projekten

In vielerlei Hinsicht stehen Open-Source-Projekte vor den gleichen Aufgaben wie andere Projekte, die etwa in einem Unternehmen durchgeführt werden; so muss beispielsweise neuer Code getestet werden. Aber es gibt auch Unterschiede zu proprietären Projekten, bei denen nur eine begrenzte Anzahl von Personen den Code ändern darf und der Quellcode nicht öffentlich zugänglich ist.

Unternehmen beauftragen in der Regel geschulte Mitarbeiter der Qualitätssicherung (Quality Assurance, QA) mit dem Testen des Codes. Viele Open-Source-Projekte bitten hingegen ihre Benutzer, jede Version zu testen. (Bei proprietären Projekten werden neben der QA auch die

Benutzer in die Tests einbezogen.).

Ein weiteres Beispiel für die Unterschiede: Ein proprietäres Projekt weist normalerweise bezahlten Entwicklern bestimmte Aufgaben zu und gibt vor, wie viel Zeit sie pro Woche für die Aufgaben aufwenden sollen. Einige Open-Source-Projekte profitieren von Mitwirkenden, die von ihren Arbeitgebern bezahlt werden oder manchmal sogar vom Projekt selbst angestellt sind, und denen Aufgaben auf die gleiche Weise zugewiesen werden wie proprietären Entwicklern. Aber in den meisten Projekten kommen viele Beiträge von Freiwilligen, die die Aufgaben erledigen, wie es ihre Zeit erlaubt. Da sich das Projekt nicht darauf verlassen kann, dass alle Freiwilligen ihre Aufgaben pünktlich abschließen, kann es bei Open Source Releases zu Verzögerungen kommen, bis die Entwickler der Meinung sind, die Funktionen sind abgeschlossen; oder die Veröffentlichung erfolgt zu festen Zeiten mit den Funktionen, die zu diesem Zeitpunkt fertig sind.

Kommunikation ist sowohl für proprietäre als auch für Open-Source-Projekte von entscheidender Bedeutung, wird aber sehr unterschiedlich gehandhabt. Proprietäre Projekte versammeln oft ein Team in einem Büro und halten regelmäßige Stand-up-Meetings nach einer Entwicklungsmethodik wie Scrum ab. Da Open-Source-Projekte meist geografisch verteilt sind, sind solche Methoden selten. Stattdessen findet die Kommunikation online und asynchron statt.

Kurz gesagt, freie Software und Open-Source-Projekte erreichen oft die gleichen Ziele wie proprietäre Projekte, aber auf andere Weise.

Das Melden von Fehlern (Bugs) ist ein kritischer Teil der Softwareentwicklung mit eigenen Rollen. Die Fehlerdatenbank muss überwacht und Prioritäten müssen zugewiesen werden. Ist ein Fehler kritisch und gefährdet sogar die Sicherheit der Software, sollte er einem kompetenten Betreuer zur raschen Behebung zugewiesen werden.

Projektleiter und Community Manager sollten die Beteiligung am Projekt beobachten und feststellen, wo es Lücken gibt. Sind zu wenige Personen bereit, den Code zu testen? Fehlt es an Dokumentation? Verlassen zu viele erfahrene Projektmitglieder das Projekt, ohne ersetzt zu werden? Projektleiter und Community Manager sollten Personen rekrutieren, die die benötigten Rollen übernehmen.

Projektleiter und Community Manager bei großen Projekten erfassen oft auch Metriken, um beispielsweise festzustellen, ob wichtige Fehler zeitnah behoben werden.

Arten von Open-Source-Beiträgen

Wie bereits erwähnt, werden Personen, die Code, Dokumentation oder anderes in das zentrale Projektarchiv einbringen, als *Contributors*, *Mitwirkende* oder *Beitragende* bezeichnet. Projektmitglieder, die Beiträge genehmigen und in das Projektarchiv aufnehmen, nennt man

Committer. Einige Mitglieder übernehmen andere übliche Rollen bei der Softwareentwicklung.

Obwohl *Contributor* üblicherweise jemanden bezeichnet, der Code oder andere Teile des offiziellen Angebots eines Projekts entwickelt, trägt jeder, der ein Projekt unterstützt, in irgendeiner Weise zu dessen Erfolg bei und verdient daher Anerkennung. Zu diesen eher informellen Beiträgen gehören das Melden von Bugs, das Beantworten von Fragen im Forum, das Spenden oder Sammeln von Geldmitteln und das Bewerben des Projekts bei Außenstehenden.

Bei Open-Source-Projekten wie auch bei proprietären Projekten werden die Benutzer befragt, welche Funktionen, Leistungsverbesserungen oder anderen Änderungen sie am Projekt wünschen — und diese Benutzer leisten dadurch ebenfalls einen wichtigen Beitrag.

Arten von Open-Source-Beitragenden

Viele Communities erhalten Beiträge nicht nur von Einzelpersonen, sondern auch von Unternehmen, die ihre Mitarbeiter dafür bezahlen, dass sie zu einem Projekt beitragen, das das Unternehmen für wichtig hält.

Es kann zu Spannungen kommen, wenn bezahlte Beitragende und Ehrenamtliche zusammenarbeiten. Ehrenamtliche können sich ausgenutzt fühlen oder befürchten, dass bezahlte Mitwirkende versuchen, die Richtung des Projekts zugunsten der Interessen ihrer Arbeitgeber zu beeinflussen. Community Manager und Projektleiter müssen sicherstellen, dass alle akzeptierten Beiträge dem gesamten Projekt und seinen Benutzern zugute kommen. Ehrenamtliche müssen motiviert werden, aus persönlichen Gründen beizutragen, sei es aus Loyalität gegenüber der Community oder um ihre eigenen Wünsche an das Projekt zu verfolgen.

Manche tragen regelmäßig zu einem Projekt bei und übernehmen langfristig Verantwortung — die sogenannten *Core Member*. Aktive Communities haben auch gelegentliche Beitragende, die Fehler melden oder Support in Foren leisten, von denen aber nicht erwartet wird, dass sie Verantwortung im Projekt übernehmen.

Einige Beitragende sind Experten auf ihrem Gebiet — unabhängig davon, ob ein Unternehmen sie für die Arbeit an dem Projekt bezahlt oder nicht --, während andere Amateure oder Enthusiasten sind. Aber man muss kein Profi sein, um eine wichtige Rolle zu übernehmen; auch als engagierter Enthusiast kann man sogar Mitglied des Kernteams oder Projektleiter werden.

Die Rolle von Organisationen in Open-Source-Projekten

Obwohl viele Open-Source-Projekte von engagierten Einzelpersonen oder kleinen Gruppen von Freiwilligen ins Leben gerufen werden, suchen sie in der Regel nach Unterstützung durch Organisationen, sobald die Projekte größer werden. Diese Unterstützung kann vielfältig sein: Im

Allgemeinen leisten Organisationen diese Beiträge, weil ein Open-Source-Projekt ihre geschäftlichen Anforderungen kostengünstiger und zuverlässiger erfüllen kann als die Neuentwicklung gleicher Funktionen in proprietärem Code. Die Garantien, die eine freie oder Open-Source-Lizenz bietet, werden von vielen Organisationen geschätzt.

Manche Idealisten misstrauen Unternehmen oder großen Organisationen und würden es vorziehen, Open-Source-Projekte gänzlich von deren Einfluss fernzuhalten. Diese Einstellung ist jedoch immer weniger verbreitet. Die meisten Open-Source-Befürworter sind vielmehr der Meinung, dass Unternehmen und andere Organisationen Open-Source-Projekten eine wichtige Grundlage bieten können.

Viele Open-Source-Projekte starten in einem Unternehmen und können sogar auf proprietärem Code basieren, den das Unternehmen zu öffnen beschließt. Um Geld zu verdienen, kann das Unternehmen die Kontrolle über das Projekt behalten und es in offene und proprietäre Teile aufteilen; man spricht hier von einem *Open-Core-Modell*. Viele Projektgründer starten mit einem Open-Source-Projekt, gründen dann aber ein Unternehmen darum herum, das das Open-Core-Modell nutzen kann (oder auch nicht).

Andere Projekte versuchen sicherzustellen, dass kein einzelnes Unternehmen ihre Richtung kontrolliert. Um Unabhängigkeit zu wahren, ist es hilfreich, mehrere organisatorische Unterstützer zu gewinnen, so dass niemand stark genug ist, diktatorische Entscheidungen zu treffen.

Wie unterstützen Organisationen Open Source? Wie bereits erwähnt, stellen viele bezahlte Mitarbeiter für Projekte ein. Darüber hinaus können sie Personen einstellen, die als Freiwillige zu dem Projekt beigetragen und Fachwissen in dem Projekt entwickelt haben. Dieser Weg zur Beschäftigung ist eine wertvolle Möglichkeit für Studierende und andere freiwillige Mitarbeiter, ihre Karriere voranzutreiben.

Unternehmen, die Erweiterungen wünschen, die andere Benutzer nicht interessieren, sollten das Open-Source-Projekt nicht unter Druck setzen, diese zusätzlichen Funktionen hinzuzufügen; sie könnten vielmehr ihre Mitarbeiter dafür bezahlen, die Funktionen zu schreiben, ohne sie an das Projekt zurückzugeben.

Ein interessantes Beispiel für die möglichen Spannungen, die durch die Bedürfnisse von Unternehmen entstehen, ist das Android Betriebssystem, das auf Linux basiert und von Google für seine mobilen Geräte entwickelt wurde. Einige von Google vorgenommene Änderungen an Linux werden an die Linux Community zurückgegeben, während andere Änderungen nur in Android erscheinen. Die Linux-Entwickler lehnen manche Änderungen ab, die ihnen von Google vorgelegt werden, so wie alle Projekte entscheiden, welche Beiträge sie übernehmen.

Es gibt viele Gründe für ein Unternehmen oder eine Gruppe von Entwicklern, eine separate

Version ihres Codes zu erstellen. Im Idealfall können sie ihre Bedürfnisse durch Hinzufügen einer optionalen Bibliothek oder einer Codesequenz erfüllen, die während der Kompilierung ausgeschlossen werden kann. Aber manchmal hat eine Gruppe auch Ziele, die mit der von den Projektleitern gewählten Richtung unvereinbar sind. Wenn dann eine separate Version erstellt wird, nennt man diese einen *Fork*.

Früher galten Forks als Folge schlechter Zusammenarbeit; heute sind sie deutlich akzeptierter. Normalerweise richten diejenigen, die einen Fork erstellen, ein neues Repository ein und starten ein neues Projekt. Manche arbeiten dann sowohl weiter an dem ursprünglichen Projekt als auch an dem Fork mit.

NOTE GitHub bezeichnet mit dem Begriff “fork” das Erstellen eines Klons oder einer Kopie des Projektcodes, um separat daran zu arbeiten.

Viele Unternehmen steuern nicht nur Code bei, sondern auch finanzielle Mittel, beispielsweise für Marketing und Konferenzen. Sie können dem Vorstand beitreten und sachkundigen Rat zu Richtung und Strategie des Projekts geben.

Ein Beispiel für diese Art von “soft support” ist der Apache Webserver. Das Projekt machte schon früh einen großen Sprung nach vorn, als IBM sein Interesse bekundete. IBM-Anwälte zeigten dem Projekt, wie es sich rechtlich absicherte, und ein von IBM bezahltes Treffen half der Apache-Führung, eine stabile Organisation aufzubauen.

Um ihre Unabhängigkeit zu wahren, gründen viele Open-Source-Projekte eine gemeinnützige Stiftung oder schließen sich einer bestehenden Stiftung an. Bekannte Beispiele für Stiftungen, die Open-Source-Projekte leiten, sind die Linux Foundation, die Apache Foundation und die Eclipse Foundation.

Die Unterstützung einer Stiftung ist wertvoll, weil sie Logistik bereitstellt, mit der sich die meisten Softwareentwickler nicht befassen wollen: juristische Unterstützung wie Markenschutz, Haftungsfreistellung und Lizenzierung, Hilfe bei der Geldbeschaffung, Infrastruktur wie Fehlerdatenbanken und Websites und so weiter.

Übertragung von Rechten

Wie bei Büchern, Musik und anderen kreativen Leistungen besteht auch bei Software ein komplexes Verhältnis zwischen den Entwicklern, den Organisationen, die ihre Beiträge verbreiten, und der Öffentlichkeit. Im Wesentlichen müssen die Mitwirkenden Schritte unternehmen, um dem Open-Source-Projekt das Recht zur Nutzung und Verbreitung ihres Codes einzuräumen.

Viele Open-Source-Organisationen fordern daher ihre Mitwirkenden auf, ein *Contributor License*

Agreement (CLA) zu unterzeichnen. Manchmal überträgt der Mitwirkende den Code einfach auf das Projekt. Das Projekt ist dann Eigentümer des Codes und aller Rechte daran, so wie ein Unternehmen die Rechte an dem Code hat, für dessen Entwicklung es seine Mitarbeiter bezahlt hat.

Andere CLAs lassen einige Rechte in den Händen der einzelnen Mitwirkenden. Viele Contributor schätzen diese Flexibilität, weil sie denselben Code zu einem anderen Projekt beitragen oder ihr eigenes Unternehmen darauf aufbauen können.

Um die unterschiedlichen Beiträge verschiedener Personen in Einklang zu bringen, ist die dem Code zugewiesene Lizenz von entscheidender Bedeutung. Der Linux Kernel ist eines der Projekte, bei dem die Eigentumsrechte in den Händen der Mitwirkenden verbleiben, so dass mittlerweile viele Tausende von Personen Rechte am Linux Code besitzen. Aber alle Beiträger zum Core Code haben diesen unter die GNU General Public License (Version 2) gestellt, so dass Linux von allen frei verwendet, verändert und weitergegeben werden kann.

Der Einsatz einer einzigen Lizenz für den gesamten Code in einem Projekt ist der einfachste Weg, Verbreitung und Verwendung des Codes einfach zu gestalten. Aber manchmal erlaubt ein Projekt, dass verschiedene Teile des Codes unter verschiedenen Lizenzen stehen, normalerweise weil das Projekt existierenden Code nutzen möchte, der bereits unter einer anderen Lizenz veröffentlicht wurde. Lizenz-Experten sollten dann sicherstellen, dass die Lizenzen kompatibel sind.

Regeln und Policies

Viele Online Communities stehen in dem Ruf, hemmungslos zu diskutieren (nach dem Motto “alles ist erlaubt”), bis hin zu Beschimpfungen und Streitereien. Heute gehen die meisten Open Source Communities dagegen vor, wenn sich Teilnehmer dazu hinreißen lassen. Vielmehr wollen moderne Communities konstruktive, höfliche und respektvolle Teilnehmer, die alle Geschlechter, ethnischen Gruppen, Persönlichkeitstypen usw. einbeziehen.

Die Erwartungen an Mitglieder werden in der Regel ausdrücklich in einem *Code of Conduct* (*Verhaltenskodex*) formuliert, der festschreibt, wie man mit anderen Personen im Projekt umgeht, sowohl online als auch persönlich. Um wirkungsvoll zu sein, muss dieser Verhaltenskodex vom Community Manager und den Projektleitern durchgesetzt werden.

Manchmal wird eine Person, die ein anderes Mitglied der Community verhöhnt oder beleidigt, dauerhaft oder vorübergehend ausgeschlossen. In anderen Fällen gibt der Community Manager oder ein Kollege öffentlich bekannt, dass das Verhalten gegen den Verhaltenskodex verstößt, und spricht möglicherweise außerhalb des Forums mit dem Täter. Oft hat dieser zuvor Frustration, mangelnde Aufmerksamkeit oder ein Burn-out erlebt, und die Mitglieder der Community können ihm helfen, besser zu kommunizieren.

Neben dem sozialen Verhalten legen Communities auch Qualitätsstandards fest. Die formellsten sind die *Coding Guidelines* (Kodierrichtlinien), die sicherstellen, dass der gesamte Code ähnlich aussieht. Die Richtlinien können Entwickler an best practices erinnern, wie beispielsweise die Verwendung von Klammern um Codeblöcke, oder Details festlegen, wie etwa die Benennung von Variablen und die Art der Einrückung.

Open Source Communities haben auch Regeln für die Freigabe von Code oder anderen Ergebnissen. Einige Projekte definieren feste Termine für die Freigabe; Ubuntu beispielsweise verspricht alle zwei Jahre eine neue Version mit Long Term Support (LTS) sowie Zwischenversionen in kürzeren Abständen. Andere Projekte führen eine Liste mit neuen Funktionen und Fehlerkorrekturen, die sie in der nächsten Version haben wollen, und geben die Version frei, wenn alles auf der Liste abgehakt ist. Aber anders als die meisten Unternehmen setzen Open Source Communities ihren Teilnehmern normalerweise keine Fristen, weil man von Freiwilligen nicht zu viel verlangen darf.

Zuschreibung und Transparenz

Neben den bereits erwähnten CLAs bitten einige Projekte die Mitwirkenden, ein *Developer Certificate of Origin* (DCO), also ein Entwicklerzertifikat, zu unterschreiben. Darin versichert der Mitwirkende, über den Code verfügen und ihn in das Projekt einbringen zu dürfen.

Warum ist das DCO wichtig? Stellen Sie sich folgendes Szenario vor: Ein Programmierer entnimmt Code aus einem proprietären Produkt seines Arbeitgebers und bringt ihn in ein Open-Source-Projekt ein; damit verletzt er die Lizenz seines Arbeitgebers und setzt das Open-Source-Projekt einem rechtlichen Risiko aus.

Das DCO soll sicherstellen, dass der Mitwirkende den Code tatsächlich geschrieben oder ihn legal erhalten hat. Das Open-Source-Projekt hängt von der Ehrlichkeit des Mitwirkenden ab, der das Zertifikat ausstellt.

Vielfalt, Gleichberechtigung, Inklusivität und Nichtdiskriminierung

Sozialwissenschaftler behaupten, dass Projekte und Unternehmen von großer Vielfalt bei Geschlecht, Ethnie, wirtschaftlichem Hintergrund, Nationalität und Fähigkeiten ihrer Mitglieder bzw. Mitarbeiter profitieren. Organisationen haben allerdings die Tendenz, sich mit anderen Menschen zu verbinden, die ihrer eigenen aktuellen Mitgliederstruktur ähnlich sind, so dass eine Community, die Vielfalt und Gleichberechtigung schätzt, ihre Mitglieder bewusst darin schulen muss, offener für Menschen zu sein, die nicht so sind wie sie selbst. Die Bewegung zur Umsetzung dieses Ideals wird *Diversity, Equity, Inclusion* (DEI) (Vielfalt, Gleichheit, Inklusion) genannt.

Der Code of Conduct ist der Ausgangspunkt für DEI. Er sollte ausdrücklich Menschen unterschiedlichen Geschlechts, unterschiedlicher Ethnien usw. willkommen heißen. Jeder Verstoß gegen den Verhaltenskodex muss umgehend und unmissverständlich missbilligt werden. Manche Minderheiten haben ihr ganzes Leben lang unter Ausgrenzung und negativen Kommentaren gelitten, und eine einzige schlechte Interaktion in einem Forum könnte dazu führen, dass sie keinen Grund mehr sehen, sich zu engagieren. Eine rasche Reaktion auf Aggressionen kann ihnen die Gewissheit geben, dass die Gemeinschaft hinter ihnen steht.

DEI geht aber noch viel weiter: Die Community sollte sich an Gruppen wenden, die sie stärker vertreten sehen will. Die Entwicklung einer App für die Stellensuche beispielsweise sollte sicherstellen, dass sie sowohl Stellen für Menschen mit niedrigem Einkommen als auch für Menschen aus der Ober- und Mittelschicht anzeigt. Die App sollte auch in einem Umfeld mit eher niedrigem Einkommen beworben werden und dort Mitglieder zum Beispiel zum Testen finden.

Eine Community könnte davon profitieren, Organisationen zu finden, die Menschen aus marginalisierten Bevölkerungsgruppen ausbilden, und dort neue Mitglieder rekrutieren. Viele solcher Organisationen sind lokal tätig und auf nationaler oder internationaler Ebene nicht bekannt.

Es ist wichtig, die Bedürfnisse von Randgruppen zu verstehen. Ist die Projektwebsite für Sehbehinderte zugänglich? Liegt Dokumentation in Sprachen vor, mit denen die Zielgruppen vertraut sind? Sollte es spezielle Foren in anderen Sprachen geben?

Die meiste Kommunikation in Open Source Communities findet asynchron und online statt. Bei Meetings oder Chat-Sitzungen sollten die Standorte/Zeitzonen der Teilnehmer berücksichtigt werden. Wenn Menschen aus vielen Ländern und Regionen auf Englisch kommunizieren, sollten Vokabular und Grammatik einfach gehalten werden.

Sind Angehörige einer Minderheit in einem Team, sollte deren Meinung gehört werden. Ein bekanntes Symptom für Ausgrenzung ist das Ignorieren der Aussagen von Minderheiten oder die Geringschätzung ihrer Bedeutung.

Andererseits sollten Mitglieder von Minderheiten nicht mit der Notwendigkeit belastet werden, die Bedürfnisse ihrer Gemeinschaften zu erklären — jeder sollte mit dieser Aufgabe betraut werden. Mitglieder von Minderheiten könnten wertvolle Aufklärungsarbeit leisten, ohne dabei unter Druck gesetzt zu werden. Jeder sollte die Möglichkeit haben, sich so zu beteiligen, wie er möchte, ohne eine Alibi-Minderheit repräsentieren zu müssen.

Geführte Übungen

1. Warum geben nicht alle Contributors ihren Code an das Projekt ab und verzichten auf alle Rechte an dem Code?

2. Wenn jemand ein Projekt unterstützen möchte, aber nicht programmieren kann, welche anderen Möglichkeiten der Unterstützung gibt es?

3. Warum schließen sich viele Open-Source-Projekte einer Stiftung an?

Offene Übungen

1. Sie sind ein Committer für ein Projekt. Jemand reicht Code ein, der von einem anderen Projekt übernommen wurde, aber der Beitragende hat die Rechte daran. Der Code ist völlig anders formatiert als der Rest des Projektcodes. Wie gehen Sie vor?

2. Sie haben ein proprietäres Softwareprodukt entwickelt und möchten Teile davon in ein Open-Source-Projekt einbringen. Unter welchen Bedingungen können Sie Ihr proprietäres Produkt weiterhin anbieten?

3. Zwei Personen auf Ihrer Mailingliste beginnen einen Streit über eine Funktion in Ihrem Code. Der Streit wird immer hitziger, bis eine Person die andere als Idioten bezeichnet. Wie können Sie mit dieser Situation umgehen?

Zusammenfassung

In dieser Lektion ging es darum, Teil einer Community zu sein und Communities produktiv zu halten. Sie haben verschiedene Arten von Beiträgen, Mitwirkenden und Rollen kennengelernt und wie Communities das Recht zur Nutzung von Beiträgen kontrollieren. Sie haben Regeln in einer Community kennengelernt und erfahren, wie sie alle Mitglieder gleichermaßen schützen.

Antworten zu den geführten Übungen

1. Warum geben nicht alle Contributors ihren Code an das Projekt ab und verzichten auf alle Rechte an dem Code?

   Ein Contributor möchte vielleicht denselben Code zu einem anderen Projekt beisteuern oder ein auf dem Code basierendes Produkt veröffentlichen und daher einige Rechte behalten.

2. Wenn jemand ein Projekt unterstützen möchte, aber nicht programmieren kann, welche anderen Möglichkeiten der Unterstützung gibt es?

   Neben dem Programmieren gibt es viele weitere Aufgaben für Mitwirkende, beispielsweise Dokumentation, Testen und Bug Reports, Community Management, Teilnahme in Foren, Werbung für das Projekt und Grafikdesign.

3. Warum schließen sich viele Open-Source-Projekte einer Stiftung an?

   Eine Stiftung übernimmt viele rechtliche, finanzielle und andere Aufgaben, die die Community des Projekts nur schwer erledigen kann.

Antworten zu den offenen Übungen

1. Sie sind ein Committer für ein Projekt. Jemand reicht Code ein, der von einem anderen Projekt übernommen wurde, aber der Beitragende hat die Rechte daran. Der Code ist völlig anders formatiert als der Rest des Projektcodes. Wie gehen Sie vor?

   Die Kodierungsstandards Ihres Projekts sollten eindeutig vorgeben, wie der Code zu formatieren ist. Danken Sie dem Mitwirkenden, verweisen Sie ihn auf die Standards und bitten Sie ihn, den Code neu zu formatieren — es kann auch Werkzeuge zur Umformatierung geben. Wenn der Mitwirkende keine Zeit hat, die Neuformatierung vorzunehmen, suchen Sie nach einem jüngeren Mitglied Ihres Teams, das diese Aufgabe übernehmen kann.

2. Sie haben ein proprietäres Softwareprodukt entwickelt und möchten Teile davon in ein Open-Source-Projekt einbringen. Unter welchen Bedingungen können Sie Ihr proprietäres Produkt weiterhin anbieten?

   Die Antwort hängt vom Contributor License Agreement ab. Wenn das CLA von Ihnen verlangt, den Code an das Projekt zu übergeben und ihm alle Rechte einzuräumen, können Sie Ihr proprietäres Produkt möglicherweise nicht weiter anbieten. Wenn Sie jedoch die Rechte an dem Code behalten dürfen, können Sie ihn unter jeder beliebigen Lizenz in Ihrem eigenen Produkt veröffentlichen.

3. Zwei Personen auf Ihrer Mailingliste beginnen einen Streit über eine Funktion in Ihrem Code. Der Streit wird immer hitziger, bis eine Person die andere als Idioten bezeichnet. Wie können Sie mit dieser Situation umgehen?

   Jeder, der die Eskalation und den beleidigenden Kommentar bemerkt, sollte so schnell wie möglich reagieren. Der Community Manager ist letztendlich für die Behebung des Schadens verantwortlich. Die Person, die eingreift, sollte einen Kommentar an die gesamte Liste schicken, dass das Verhalten gegen den Code of Conduct des Projekts verstößt (der hoffentlich ein solches Verhalten ausschließt). Die Person, die eingreift, kann auch die Streitenden einzeln ansprechen, um sicherzustellen, dass sie mit der Lösung des Streits zufrieden sind und verstehen, wie man Meinungsverschiedenheiten konstruktiv diskutiert.

Linux Professional Institute

Thema 056: Zusammenarbeit und Kommunikation

056.1 Werkzeuge der Softwareentwicklung

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 056.1

Gewichtung

2

Hauptwissensgebiete

- Verständnis der gängigen Werkzeuge der Softwareentwicklung
- Verständnis der gängigen Deployment-Umgebungen
- Verständnis der gängigen Arten von Softwaretests
- Verständnis der Konzepte von Continuous Integration und Continuous Delivery (CI/CD)

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Integrated Development Environments (IDEs)
- Linter
- Compiler
- Debugger
- Unit-Tests
- Integrationstests
- Abnahmetests
- Leistungstests
- Smoke-Tests
- Regressionstests

- Produktions-, Staging- und Entwicklungssysteme
- Lokale Entwicklungssysteme
- Remote-Entwicklungssysteme
- CI/CD-Pipelines

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	056 Zusammenarbeit und Kommunikation
Lernziel:	056.1 Werkzeuge der Softwareentwicklung
Lektion:	1 von 1

Einführung

Es gibt Tausende Werkzeuge zur Softwareentwicklung — sowohl Open Source als auch proprietär. Warum auch nicht? Programmierer lieben es, Werkzeuge für sich selbst und ihre Kollegen zu entwickeln; es ist nur natürlich, Zeit in die Entwicklung von Tools zu investieren, die besser funktionieren, Ärgernisse im Arbeitsablauf beseitigen oder die Bereitstellung erleichtern.

Diese Lektion konzentriert sich auf den *Prozess* der Softwareentwicklung und erklärt, welche Rolle verschiedene Arten von Entwicklungswerkzeugen in diesem Prozess spielen. Es werden nur wenige spezifische Werkzeuge genannt; möglicherweise gelten manche bereits als überholt und wurden durch neue Favoriten ersetzt, wenn Sie diese Lektion lesen.

Ziele der Entwicklung

Programmierwerkzeuge haben verschiedene Ziele, die manchmal zueinander in Konflikt stehen. Hier einige Beispiele:

- Programme erzeugen, die stabil und korrekt sind
- Programme erzeugen, die schnell laufen

- Programme erzeugen, die gut skalieren
- Programme erzeugen, die sich an verschiedene Benutzertypen, Geräte (Laptops, Handys, Tablets etc.) und Umgebungen anpassen lassen
- Entwicklungsprozess beschleunigen
- Bereitstellung neu entwickelter Funktionen oder Fehlerkorrekturen beschleunigen
- Monotone Arbeitsabläufe und die Anzahl von Programmierfehlern reduzieren, die sich in den Testphasen des Programms einschleichen
- Legacy Code und Geräte unterstützen, die ein Unternehmen nur schwer ersetzen kann
- Mit anderen bekannten Tools zusammenarbeiten
- Einfaches Rollback im Falle von Fehlern oder Planänderungen erlauben

Diese und andere Ziele führen zu den in dieser Lektion beschriebenen Prozessen und den daraus resultierenden Entwicklungswerkzeugen.

Allgemeine Entwicklungsprozesse

In diesem Abschnitt werden zwei wichtige Modelle gegenübergestellt: das *Wasserfallmodell* (das bereits in einer früheren Lektion vorgestellt wurde) und *Continuous Integration/Continuous Delivery* (CI/CD).

Wasserfallmodell

Obwohl das Wasserfallmodell vor allem in großen Unternehmen immer noch weit verbreitet ist, gilt es vielen Entwicklern heute als überholt. Das Modell war von den 1950er bis in die 1970er Jahre populär. Elemente des Modells sind auch heute noch aktuell, wie beispielsweise die Formalisierung von Anforderungen und das Testen von Programmen kurz vor ihrer Freigabe (*Qualitätssicherung*).

Schon zu der Zeit, als der Begriff “Wasserfall” für dieses Modell geprägt wurde, war es durch eine Reihe von Problemen in Verruf geraten:

- Nach Beginn der Entwicklung war es schwierig, die Anforderungen zu ändern.
- Anforderungen und Entwürfe wurden oft missverstanden, da Texte in einfacher Sprache mehrdeutig sein konnten. Das führte zu Produkten, die nicht den beabsichtigten Anforderungen entsprachen und deren Korrektur Monate in Anspruch nahm.
- Der Prozess war langsam: Eine neue Version erschien vielleicht nur einmal im Jahr oder sogar noch seltener.

- Fehler, die durch das Zusammenspiel verschiedener Module entstanden, waren erst spät zu erkennen, was zu weiteren Verzögerungen führte.
- Der Prozess schuf Barrieren zwischen den verschiedenen Teilen der Organisation, was sich sowohl auf die Qualität des Produkts als auch auf den Zusammenhalt in der Organisation negativ auswirkte.

Grundsätze von Continuous Integration/Continuous Delivery (CI/CD)

Das Wasserfallmodell wurde in den 1970er Jahren zunehmend in Frage gestellt, was zu dem heute führenden (wenn auch nicht allgemein verwendeten) Modell führte: CI/CD. Wichtige Etappen auf dem Weg zu den CI/CD-Methoden waren das 2001 veröffentlichte *Agile Manifest*, Scrum und DevOps. Das neue Modell basiert auf der intensiven Kommunikation zwischen den Teammitgliedern, der Einbeziehung der Benutzer oder Kunden, der schnellen Umsetzung von Fehlerkorrekturen und neuen Funktionen sowie konsequent fortlaufenden Tests, um die Qualität in einer sich schnell verändernden — manchmal sogar chaotischen — Umgebung zu erhalten.

CI/CD automatisiert in jeder Phase so viele Schritte wie möglich, vom Schreiben des Codes bis zur Bereitstellung des fertigen Programms für die Benutzer. CI/CD erfordert, dass jeder Schritt eindeutig definiert und menschliches Handeln (wie die Installation von Software) durch einen von einem Programm ausführbaren Vorgang ersetzt wird. Somit ist CI/CD Teil einer Bewegung, die unter den Slogans “Alles als Code” und “Infrastruktur als Code” bekannt ist.

Sobald ein Team seine Prozesse in den Code integriert hat, können diese Prozesse ebenso wie der Code korrigiert, aktualisiert und aufgezeichnet werden. Alles, was das Team schreibt — seien es Programme, automatisierte Verfahren oder Dokumentation — sollte in einem Versionskontrollsystem gespeichert werden, das in einer der nächsten Lektionen beschrieben wird.

Werden mehrere Prozeduren automatisiert, können sie nacheinander ablaufen. Daher sprechen Teams oft von einer *CI/CD Pipeline*: ein erfolgreicher Schritt in der Pipeline löst automatisch einen oder mehrere nachfolgende Prozesse aus. Eine Pipeline muss automatisiert überwacht werden, so dass jeder Fehler auf dem Weg dazu führt, dass die Pipeline angehalten und Teammitglieder über den Fehler informiert werden.

In den folgenden Abschnitten werden CI und CD zwar getrennt beschrieben, doch werden sie häufig miteinander kombiniert, und die Grenzen zwischen ihnen sind fließend.

Continuous Integration (CI)

Continuous Integration beschreibt die schnelle Einarbeitung kleiner Änderungen in den Code. Das zentrale Repository, in dem das Produkt für die Benutzer entsteht, wird als *Core Repository* (oder

Core Repo) bezeichnet. Jeder Programmierer erstellt einen persönlichen Arbeitsbereich (oft *Sandbox* genannt) auf seinem Computer und zieht die Dateien, die er korrigieren oder aktualisieren muss, aus dem Core Repository.

Der Programmierer kann einen persönlichen Laptop oder Desktop verwenden (als *Local Development System*), relevante Teile des Core Repository herunterladen und die Änderungen nach lokalen Tests hochladen. Alternativ kann er die Vorteile des Cloud Computing nutzen und auf einem gemeinsam genutzten System arbeiten, das von seiner Organisation als *Remote Development System* betrieben wird.

Der Programmierer prüft in der Regel mit einem *Debugger*, einem separaten Programm, das die Arbeit des Programmierers in einer kontrollierten Umgebung ausführt, auf Fehler. Debugger und andere Werkzeugen zum Auffinden von Fehlern werden später noch Thema sein.

Um Fehler so früh wie möglich im Entwicklungsprozess zu erkennen, führt der Programmierer auch Tests am Code durch, bevor er ihn in das Core Repository hochlädt. Diese werden *Unit Tests* genannt, weil sich jeder Test auf ein kleines Element des Codes konzentriert: Hat beispielsweise eine Funktion einen Zähler erhöht, wie es vorgesehen war?

In der letzten Phase von CI werden die Änderungen des Programmierers in das Core Repository eingepflegt. Dann führen die Tools *Integrationstests* durch, um sicherzustellen, dass der Programmierer nichts im Gesamtprojekt kaputt gemacht hat.

Unabhängig davon, wie weit die Automatisierung fortgeschritten ist, sollte ein fachkundiges Teammitglied die Integration überwachen, um sicherzustellen, dass die Änderung vom Team gewünscht wird. Automatisierte Tests können feststellen, dass nichts kaputt gegangen ist, und sogar, ob die Änderung den gewünschten Effekt im Programm erzeugt. Aber diese Tests können nicht auf alles prüfen, was das Team für wichtig hält.

Bevor ein Teammitglied mit dem Deployment, also der Bereitstellung, beginnt, sollten die Sicherheit, die Einhaltung von Kodierungsstandards, die ordnungsgemäße Dokumentation und andere Grundsätze überprüft werden — und es überrascht nicht, dass es auch für einige dieser Aufgaben wiederum eigene Tools gibt.

Während der CI finden viele Tests statt, die schnell und zuverlässig ablaufen müssen, um das Tempo der modernen Entwicklung zu unterstützen. Daher sind moderne Tools für die Integration von Code und die Ausführung von Tests automatisiert. Ein Programmierer sollte in der Lage sein, eine ganze Suite von Unit Tests mit einem einzigen Befehl oder einem Klick auf eine Schaltfläche auszuführen.

Üblicherweise wird ein teilweiser oder vollständiger Integrationstest ausgeführt, sobald ein Programmierer Code in das Core Repository hochlädt. Alle durch die Tests festgestellten Fehler

werden dem Programmierer schnell gemeldet.

Continuous Delivery (CD)

Wie bereits erläutert, besteht CI aus automatisierten Verfahren, die zu einer neuen Version des Programms im Core Repository führen. *Continuous Delivery* bezieht sich auf alle nachfolgenden Prozesse, um die neue Version den Benutzern bereitzustellen. Das D in CD wird daher neben “Delivery” oft auch als “Development” oder “Deployment” verstanden.

Die Hauptaufgaben des CD bestehen darin, den Code gründlich zu testen und ihn auf die Computer der Benutzer oder in ein Anwendungsrepository hochzuladen.

In der CD-Phase wird eine Reihe von Tests durchgeführt, um sicherzustellen, dass das Produkt einwandfrei funktioniert. Oft wird ein größerer Satz von Integrationstests mit einer Reihe anderer Tests kombiniert, um die Auswirkungen für die Benutzer, die Leistung und die Sicherheit zu ermitteln. Einige dieser Tests werden in einem späteren Abschnitt beschrieben.

Die Tests sollten auf anderen Systemen als den Produktionssystemen erfolgen, die die Benutzer bedienen. Ein schwerwiegender Fehler könnte nicht nur das Produktionssystem zum Absturz bringen, sondern auch die Benutzerdaten kompromittieren. Außerdem benötigen die Tests Systemzeit und beeinträchtigen die Performance für die Benutzer.

Zum Testen sind daher eigene Umgebungen sinnvoll, die die Produktionsumgebung abbilden, mit ähnlicher Hardware und Software. Solche Zwischensysteme, in denen die Tests vor der Auslieferung erfolgen, werden oft als *Staging*-Umgebung bezeichnet.

Natürlich ist es nicht praktikabel, die Testumgebung so groß wie die Produktionsumgebung anzulegen, aber die wesentlichen Elemente der Produktionsumgebung (beispielsweise Datenbanken) sollten enthalten sein.

Voraussetzung für die CD-Automatisierung ist die Unterscheidung zwischen Test- und Produktionsumgebung: Alle Code-Installationen und -Prozesse sollten auf die Zielumgebung zugeschnitten sein.

CD bietet das größte Potenzial für eine schnelle Entwicklung, wenn der Dienst auf den eigenen Systemen des Unternehmens läuft. Ein Einzelhandelsunternehmen mit einer interaktiven Webseite hat in der Regel Zugriff auf alle Webserver; es kann diese bei Bedarf mehrmals am Tag aktualisieren und Änderungen schnell zurücknehmen, wenn sich herausstellt, dass sie den Benutzern Probleme bereiten.

Werkzeuge zur Softwareentwicklung

In den folgenden Abschnitten geht es um wichtige Arten von Tools, die Programmierteams auf drei Ebenen nutzen: Code-Generierung, Testen und Deployment.

Compiler

Ein Basiswerkzeug der Programmierung ist der *Compiler*, der komplexe, sogenannten höhere oder High-Level-Programmiersprachen in Anweisungen für den Computerprozessor übersetzt.

Computer führen *Maschinencode* aus, der aus Bitfolgen (Einsen und Nullen) besteht, die der Prozessor wiederum in Befehle und Daten umwandelt: Laden eines Wertes aus dem Speicher in ein Register, Addieren der Werte zweier Register usw. Prozessoren unterscheiden sich hinsichtlich des Formats und der Befehlssätze, die ihnen zur Verfügung stehen; darum nutzen sie auch unterschiedlichen Maschinencode. Hersteller versuchen, neue Prozessoren so zu entwickeln, dass sie denselben Maschinencode nutzen, damit Kunden ihre alten Programme auch auf neuen Prozessoren ausführen können. Man spricht in diesem Fall von *Prozessorfamilien*.

Die nächste Stufe in den Anfangsjahren der Programmierung war die *Assemblersprache*, die für jede Anweisung menschenlesbare Begriffe wie beispielsweise `ADD` (für das Addieren) bereitstellte. Programmierer schrieben in Assemblersprache und übergaben ihren Code an ein Werkzeug, den *Assembler*, der die Anweisungen in Maschinencode übersetzte. Maschinencode wird übrigens auch als *Maschinensprache* bezeichnet.

Darauf folgte die Entwicklung höherer Programmiersprachen. Hier lassen sich komplexe Abläufe erstellen, ohne deren Details zu spezifizieren, indem man die gewünschten Ergebnisse bestimmt. Diese Programme benötigen einen Compiler, um den Quellcode in Maschinencode umzuwandeln. Diese Compiler können ziemlich intelligente Transformationen und Optimierungen durchführen.

Für viele Sprachen stehen mehrere Compiler zur Verfügung, beispielsweise für C und Java. In der FOSS Community kommen meist entweder die GNU Compiler Collection (GCC) oder der LLVM Compiler für C und C++ zum Einsatz.

Ursprünglich kompilierte ein Compiler jede Datei des Quellcodes und erzeugte daraus jeweils eine *Objektdatei* (*Object File*), um dann ein weiteres Programm, den *Linker*, aufzurufen, der alle Objektdateien zu einem Programm zusammenfügte. Moderne Compiler können mehrere Dateien gleichzeitig kompilieren, um Optimierungen durchzuführen, die die Grenzen von Dateien überschreiten.

Früher kompilierten Compiler in Assemblersprache, was nützlich sein konnte, da jeder Prozessortyp einen anderen Maschinencode, aber möglicherweise dieselbe Assemblersprache unterstützte. Es gibt mehrere Varianten von Assemblersprachen. Der letzte Schritt beim

Kompilieren und Linken bestand darin, Maschinencode zu erzeugen. Wie beim Linken wissen moderne Compiler, wie sie den Code in Maschinencode umsetzen.

In vielen modernen Sprachen gibt es jedoch noch eine weitere Stufe: den Zwischencode, meist als *Bytecode* bezeichnet. Dieser Code wurde in ein Binärformat kompiliert, das high-level genug ist, um portabel zu sein. Die Sprache Java wird beispielsweise in Bytecode kompiliert, so dass das Programm auf viele verschiedene Prozessortypen geladen werden kann.

Bei der Erstellung von Bytecode aus Quellcode hat der Compiler einen Großteil der Arbeit erledigt. Jeder Computer hostet dann seine eigene Version eines Programms, das als *virtuelle Maschine* bezeichnet wird und die Umwandlung von Bytecode in eine Reihe von Anweisungen vollzieht, die die virtuelle Maschine ausführen kann. Manchmal wird Bytecode auch in Maschinencode kompiliert. Der Übergang von Bytecode zu einer Reihe von Anweisungen ist für den Benutzer bequemer als der Übergang von einer höheren Programmiersprache zu Maschinencode. Dies war ein großer Vorteil für Java, als es erfunden wurde, denn seine Entwickler wollten, dass es innerhalb eines Plug-ins für eine virtuelle Maschine für Webbrowser läuft, wo der Prozessor und die Betriebsumgebung des Benutzers sehr unterschiedlich sein können.

Die Java-Entwickler bewarben die Vorteile von Bytecode mit dem Slogan “compile once, run anywhere”. Später ergaben sich noch weitere Vorteile von Bytecode: Es ließen sich neue Sprachen mit ganz anderen Ansätzen für Programmierer entwickeln (was die Arbeit einfacher und zu besser wartbarem Code führen sollte), die denselben Bytecode erzeugten, so dass er von den Java Virtual Machines unterstützt wurde. Funktionen aus diesen Sprachen ließen sich leicht in bestehende Java-Programme einfügen.

Schließlich gibt es einige Sprachen wie Python, für die man den Quellcode überhaupt nicht kompilieren muss: Man gibt die Anweisungen einfach in ein Verarbeitungsprogramm ein, das als *Interpreter* bezeichnet wird und das den Code direkt in Maschinencode umwandelt und ausführt. Interpreter sind langsamer als Compiler, aber einige sind inzwischen so effizient, dass sie keine großen Leistungseinbußen mit sich bringen. Dennoch enthalten einige beliebte Bibliotheken in der Sprache Python Funktionen, auf die Entwickler in der interpretierten Sprache zugreifen können, die aber in der Sprache C programmiert sind, um die Ausführung zu beschleunigen.

Für viele interpretierte Sprachen gibt es auch Compiler, die entweder Byte- oder Maschinencode erzeugen, der dann schneller ausgeführt werden kann als von einem Interpreter.

Es gibt Build-Tools, die dem Programmierer bei der Verwaltung von Dateien und Funktionen helfen. Ein komplexes Programm kann Hunderte von Dateien enthalten, und der Programmierer muss verschiedene Kombinationen von Dateien mit verschiedenen Compiler-Optionen zu verschiedenen Zeitpunkten kompilieren (beispielsweise zur Unterstützung von Debugging). In einem Build-Tool kann man verschiedene Optionen und Kombinationen von Dateien speichern

und einfach den gewünschten Build-Typ auswählen. Maven und Gradle sind gängige Build-Tools für Java und verwandte Sprachen.

Werkzeuge zur Code-Generierung

Programmierer starten nicht mit einem leeren Bildschirm. Es gibt inzwischen zahlreiche Tools, die Code auf der Grundlage von Klartextbeschreibungen generieren. Wie bei anderen Formen generativer KI wird dabei häufig kritisiert, dass sie auf der Arbeit anderer Programmierer basiert ohne diese zu honorieren und dass der erzeugte Quellcode weniger robust sei. Abgesehen von ethischen Kontroversen stellen zahlreiche Programmierer aber auch fest, dass die automatische Code-Generierung ihre Produktivität erheblich verbessert hat.

Eine Form der Code-Generierung, die seit vielen Jahren in vielen Programmiersprachen verfügbar ist, ist das *Refactoring*. Dabei wird ein großes Programm, das sich im Laufe der Zeit zu einem Gewirr von Dateien und Funktionen entwickelt hat, analysiert und so neu arrangiert, dass eine logischere Struktur entsteht, sich die Wartbarkeit verbessert und sich Doppelungen im Quellcode reduzieren.

Manche Programmierer haben die Aufgabe, die Funktionsweise von fremdem Code zu reproduzieren. Sie müssen vielleicht ein neues Programm schreiben, um ein altes Programm zu ersetzen, dessen Quellcode nicht mehr zur Verfügung steht oder ausgemustert werden muss. Oder sie ahmen das Programm eines Konkurrenten nach. Diese Art von Analyse wird als *Reverse Engineering* bezeichnet. Ein nützliches Werkzeug für diesen Zweck ist ein *Disassembler*, der Maschinencode in Assembler umwandelt. Es gibt auch Disassembler für Bytecode.

Debugger

Die meisten Programmierer verbringen mehr Zeit mit der Fehlersuche als mit dem Programmieren. Menschen denken einfach nicht vollkommen logisch und vergessen daher oft Details, die der Computer benötigt, um das Programm wunschgemäß auszuführen. Daher ist es wahrscheinlich, dass Code beim ersten Versuch scheitert; ein *Debugger* hilft, die Fehler aufzudecken.

Ein Debugger unterstützt intensive Analysen, die den Prozess der Fehlersuche um Stunden verkürzen können.

Ein Programmierer kann das Programm so schreiben, dass es an einer Schlüsselstelle (einem *Breakpoint*) stoppt, beispielsweise zu Beginn einer Funktion oder Schleife. Der Debugger zeigt die Werte von Variablen und sogar Computerregistern an der aktuellen Stelle im Programm an. Der Programmierer kann auch jede Anweisung einzeln ausführen und die Ergebnisse sehen (*Single Stepping*) oder einen *Watchpoint* setzen, um nachzuverfolgen, wann und wie sich eine Variable

während des Programmablaufs ändert.

Der bekannteste Debugger in der FOSS-Welt, insbesondere für C und C++, ist der GNU Debugger, der mit dem bereits erwähnten GNU Compiler zusammenarbeitet. Auch für andere Sprachen gibt es spezielle Debugger.

Analysetools

Obwohl Fehler durch Debugging in der Regel recht schnell entdeckt werden, empfiehlt es sich, Rechtschreibfehler und andere grundlegende Probleme bereits in einem früheren Stadium des Programmierprozesses zu beseitigen. Es gibt viele ausgeklügelte Analysewerkzeuge, um ein Programm zu prüfen. *Statische Analyse* untersucht den Programmcode, *dynamische Analyse* führt ein Programm aus und prüft es während seiner Ausführung auf Probleme.

Eines der frühesten Tools zur statischen Analyse wurde als *Linter* bezeichnet. Dieser stellt beispielsweise fest, ob der Wert einer Fließkommavariablen einer Ganzzahl zugewiesen wurde. Dies kann zu Problemen führen — oder auch nicht; und es kann vom Compiler erkannt werden — oder auch nicht. Im produktiven Einsatz kann es daher zu falschen Ergebnissen führen.

Heute übernehmen die meisten Compiler die Aufgabe eines Linters. Einige Compiler, wie der für die Sprache Rust, zeichnen sich dadurch aus, dass sie schlecht geschriebenen Code strikt ablehnen.

Viele andere Arten von Analysewerkzeugen laufen unabhängig vom Compiler. Tools zur Sicherheitsanalyse spüren beispielsweise Probleme auf, die ein Programm anfällig für Hacker machen. Ein häufiger Programmierfehler besteht zum Beispiel darin, dass Code eine Funktion aufruft und nicht überprüft, ob diese Funktion einen Fehler zurückgibt.

Integrated Development Environments (IDEs)

Viele Programmierer nutzen Texteditoren, um ihren Code zu schreiben und zu bearbeiten. Texteditoren unterscheiden sich von Textverarbeitungsprogrammen, die zahlreiche Funktionen zur Formatierung (wie kursiv und fett, Aufzählungszeichen und nummerierte Listen usw.) mitbringen. Texteditoren erzeugen blanken Text, den Programmiersprachen erfordern.

Es gibt aber auch hochentwickelte Tools, die die Softwareentwicklung unterstützen. Diese Tools sind auf die verwendete Programmiersprache eingestellt. Gibt man beispielsweise den Namen einer Variablen oder Funktion ein, kann das Tool eine Vervollständigung vorschlagen. Diese Tools können auch bereits während des Programmierens nach Fehlern suchen, den Code einheitlich und ansprechend formatieren, Analysewerkzeuge und Debugger ausführen, den Code kompilieren, Check-Ins in Versionskontrollsysteme durchführen und andere Aufgaben übernehmen. Daher werden sie auch als *Integrated Development Environments* (IDE), also

integrierte Entwicklungsumgebungen, bezeichnet.

Eclipse ist eine beliebte Open-Source-IDE.

Arten von Softwaretests

Das Testing ist ein wichtiger Teil der Softwareentwicklung. Programmierer führen Unit Tests durch, während sie den Code entwickeln. Andere Arten von Tests erfolgen typischerweise, wenn Code zurück ins Core Repository übertragen wird, oder beim Deployment.

Unit Testing

Wir haben gesehen, dass Programmierer unbedingt Fehler finden sollten, bevor sie Code zur Integration in das Core Repository einreichen. Unit Tests sind entscheidend bei der Fehlersuche.

Das Schreiben dieser Tests ist Kunst und Wissenschaft zugleich, und der Umfang des Testcodes kann den des produktiven Codes übersteigen. Es gibt sogar ein Entwicklungsmodell namens *Test-Driven Development* (TDD), bei dem Programmierer zunächst Tests schreiben, bevor sie den zu testenden Code entwickeln. Dieses Vorgehen soll Lücken beim Testen vermeiden und dazu beitragen, dass der Code seine Aufgabe verlässlicher ausführt.

Es ist wichtig, sowohl das zu testen, was während der Programmausführung schief geht, als auch das, was korrekt läuft. Wenn ein Benutzer oder ein anderer Teil des Programms einer Funktion eine ungültige Eingabe übergibt, ist es wichtig, dass die Funktion das Problem erkennt und eine entsprechende Fehlermeldung ausgibt.

JUnit ist ein beliebtes Open Source Tool zur Durchführung von Unit Tests für Java-Programme.

Integrations-, Regressions- und Smoke Tests

Während sich Unit Tests auf einzelne Aktionen bestimmter Funktionen konzentrieren, sollte das Team auch Tests auf einer höheren Ebene durchführen, um sicherzustellen, dass das Produkt als Ganzes ordnungsgemäß funktioniert. Die Tests imitieren in der Regel das Benutzerverhalten. In einer Anwendung für das Restaurantmanagement könnten die Tests beispielsweise prüfen, ob ein Benutzer den gewünschten Artikel erhält.

Wenn eine Änderung an einem Programm eine Funktion unterbricht, die vorher funktionierte, nennt man den Fehler eine *Regression*, und die Tests entsprechend *Regressionstests*.

Wenn ein Team ein Produkt zur Freigabe vorbereitet, ist die erste Testphase oft sehr kurz. Das Team prüft die wichtigsten Abläufe, die ein Programm ausführt, und bricht den Test ab, sobald Fehler auftauchen, um Zeit zu sparen. Diese Art des Testens wird als *Smoke Test* bezeichnet, denn

eine Anwendung, die derart fehlerhaft ist, gleicht einem defekten Gerät, das Feuer fängt.

Einige Produkte erfordern Benutzerinteraktion, beispielsweise Web- und mobile Anwendungen. Der Test läuft automatisch ab und löst den Code aus, der ausgeführt worden wäre, wenn ein Benutzer die Schaltfläche gedrückt hätte. Selenium ist ein beliebtes Tool in dieser Kategorie.

Abnahmetests

Anwendungen mit einer Benutzeroberfläche benötigen eine zusätzliche Testebene, die nicht nur prüft, ob sie auf bestimmte Eingaben richtig reagieren, sondern auch, ob sie auf dem Bildschirm korrekt dargestellt werden. *Acceptance Tests* (also Akzeptanz- oder Abnahmetests) prüfen die Wirkung des Programms auf den Benutzer. Teams zur Qualitätssicherung führen diese Tests in der Regel durch, nachdem Integrations- und Regressionstests gezeigt haben, dass das Programm formal korrekt ist und die Erwartungen der Benutzer erfüllt.

Sicherheitstests

Sicherheit ist natürlich von entscheidender Bedeutung, denn selbst das kleinste Sicherheitsversagen kann einem Unternehmen großen Schaden zufügen. Wir haben gesehen, dass Analysewerkzeuge die Sicherheit des Codes prüfen. In der Qualitätssicherungsphase können Tests auch feststellen, ob das Programm Schwachstellen aufweist. Die Tests führen das Programm mit bösartigen Eingaben aus und stellen sicher, dass das Programm die Eingaben zurückweist, ohne abzubrechen, unbeabsichtigte Aktionen auszuführen oder sensible Informationen preiszugeben.

Eine Art von Test, die sich in manchen Fällen als wertvoll erwiesen hat, ist das *Fuzz Testing*. Das Test-Framework generiert zufällige, sinnlose Zeichenketten und übergibt sie als Eingabe dem Programm. Das mag Zeitverschwendung scheinen, deckt aber oft Schwachstellen auf, die bei normalen Tests nicht gefunden werden.

Performancetests

Hat ein Programm andere Tests erfolgreich bestanden, sollten die Teams feststellen, ob es auch schnell genug läuft. *Performancetests* erfordern eine Umgebung, die derjenigen ähnelt, in der die Benutzer mit dem Programm interagieren werden. Senden Benutzer beispielsweise Anfragen aus großer Entfernung über ein Netzwerk, sollten die Leistungstests auch über ein solches Netzwerk durchgeführt werden.

Einige Programmierbibliotheken werden durch *Benchmarks* getestet; das sind Standardtests, die dem Vergleich verschiedener Bibliotheken oder verschiedener Versionen derselben Bibliothek dienen.

Deployment-Umgebungen

Ein CI/CD-Tool bietet ausgefeilte Möglichkeiten zum Aufbau von Pipelines. Wie Computerprogramme kann eine Pipeline Tests und Branches enthalten. Durch Branches lassen sich manche Aktionen in der Testumgebung, andere in der Produktionsumgebung durchführen. Das Tool kann beispielsweise dazu dienen, automatisch die richtige Datenbank oder andere Software zu installieren, die für verschiedene Programme benötigt wird.

CD überschneidet sich mit DevOps. In Cloud-Umgebungen erstellen CD- und DevOps-Tools automatisiert die virtuellen Computersysteme mit allen Komponenten, die für die Ausführung der Programme erforderlich sind. Automatisierte Tools (manchmal auch als Werkzeuge zur *Orchestrierung* bezeichnet) prüfen, ob die virtuellen Systeme ausfallen, und starten sie automatisch neu.

Die Hauptaufgabe eines CD-Werkzeugs besteht darin, jede Pipeline zu starten und schrittweise zu durchlaufen. Das Werkzeug prüft die Ergebnisse jeder Phase der Pipeline und entscheidet, ob es fortgesetzt oder gestoppt werden soll. Das Werkzeug ermöglicht auch die Zeitplanung und protokolliert seine Aktivitäten.

Oft muss eine Aufgabe mit geringfügigen Änderungen wiederholt werden. So unterscheiden Teams beispielsweise zwischen der Bereitstellung in einer Test- und in der Produktionsumgebung. Daher stellen CD-Tools Parameter bereit, denen beim Ausführen der Pipeline unterschiedliche Werte mitgegeben werden.

Manchmal benötigt ein Prozess Befehle, die traditionell am Terminal eingegeben wurden. Daher bietet ein CD-Werkzeug Mechanismen, um beliebige Befehle auszuführen. Normalerweise bietet es Hooks wie `preprocess`, um Befehle vor einer Phase der Pipeline auszuführen, und `postprocess`, um Befehle nach einer Phase der Pipeline auszuführen.

Jenkins ist wahrscheinlich das beliebteste Open Source Tool für die in diesem Abschnitt beschriebene Orchestrierung.

Geführte Übungen

1. Welche Möglichkeiten gibt es, die Sicherheit eines Programms zu überprüfen?

2. Warum sollten Sie mehrere Unit Tests für eine einzige Programmfunktion schreiben?

Offene Übungen

1. Ihr Team hat eine alte Anwendung übernommen, die langsam läuft und nur schwer um neue Funktionen zu erweitern ist. Welche Möglichkeiten gibt es, die Anwendung zu verbessern, ohne sie zu verwerfen und eine neue Anwendung von Grund auf zu schreiben?

2. In großen Projekten kommt es häufig vor, dass Team A eine Funktion in einem Teil des Systems implementieren möchte, der von Team B gepflegt wird, Team B die Funktion jedoch nicht als vorrangig ansieht. Wie kann Team A die Funktion als Teil des Projekts von Team B kodieren?

Zusammenfassung

In dieser Lektion wurden verschiedene Arten von Tools zur Softwareentwicklung vorgestellt: Compiler und andere Werkzeuge zur Code-Generierung, für Analyse und Tests sowie CI/CD-Tools, die die Integration und Bereitstellung automatisieren. Es gibt zahlreiche Optionen für jede dieser Aufgaben, und ein Tool, das heute beliebt ist, kann in einem Jahr ersetzt werden. Wenn Sie verstehen, wie diese Tools im Entwicklungsprozess zusammenpassen, können Sie herausfinden, was Sie brauchen.

Antworten zu den geführten Übungen

1. Welche Möglichkeiten gibt es, die Sicherheit eines Programms zu überprüfen?

   Zunächst können Experten den Code prüfen.

   Darüber hinaus gibt es zahlreiche statische und dynamische Analysetools, die schlechte Programmierpraktiken aufspüren, die ein Programm anfällig für Angriffe auf die Sicherheit machen.

   Andere Tools senden bösartige Eingaben an laufende Programme und überprüfen deren Reaktionen.

2. Warum sollten Sie mehrere Unit Tests für eine einzige Programmfunktion schreiben?

   Eine Programmfunktion muss in der Regel mit vielen verschiedenen Eingabewerten laufen, so dass jede Variante einen eigenen Test verdient. Zum Beispiel könnte die Funktion den Eingabewert Null auf eine spezielle Weise behandeln. Sie müssen auch ungültige Eingaben vorhersehen und Tests schreiben, um zu zeigen, dass die Funktion sie angemessen behandelt.

Antworten zu den offenen Übungen

1. Ihr Team hat eine alte Anwendung übernommen, die langsam läuft und nur schwer um neue Funktionen zu erweitern ist. Welche Möglichkeiten gibt es, die Anwendung zu verbessern, ohne sie zu verwerfen und eine neue Anwendung von Grund auf zu schreiben?

   Vergewissern Sie sich zunächst, dass das Projekt unter Versionskontrolle steht, falls das nicht bereits der Fall ist.

   Führen Sie nach dem Hinzufügen einer neuen Funktion Regressionstests durch, um festzustellen, wo das Programm versagt, und beauftragen Sie Programmierer, herauszufinden, welche Funktionen dafür verantwortlich sind. Diese Funktionen können selektiv ersetzt werden.

   Durch Performancetests lassen sich Funktionen ermitteln, die langsam laufen, so dass Sie Ihre Bemühungen darauf konzentrieren können, den ineffizientesten Code zu korrigieren oder zu ersetzen.

   Wenn der Code in einer nicht mehr gebräuchlichen Sprache vorliegt, sollten Sie Funktionen in einer vom Team bevorzugten Sprache hinzufügen. Stellen Sie sicher, dass die Funktionen in der neuen Sprache mit den alten Funktionen integriert werden können.

2. In großen Projekten kommt es häufig vor, dass Team A eine Funktion in einem Teil des Systems implementieren möchte, der von Team B gepflegt wird, Team B die Funktion jedoch nicht als vorrangig ansieht. Wie kann Team A die Funktion als Teil des Projekts von Team B kodieren?

   Team B kann Team A erlauben, einen neuen Branch zu erstellen, die Funktion zu kodieren und den Branch an Team B zu übermitteln, damit es ihn in sein Projekt einbindet. Team A darf jedoch nicht alles tun, was es will. Team B sollte Dokumentation und Hilfe für Team A bereitstellen, um die Projektstandards zu befolgen. Ein Mitglied von Team B muss außerdem die Beiträge von Team A überprüfen und alle üblichen Integrationstests durchführen.

   Diese Form der Zusammenarbeit wird manchmal als InnerSource bezeichnet, weil sie Open Source ähnelt, aber innerhalb einer einzigen Organisation stattfindet.

056.2 Source Code Management

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 056.2

Gewichtung

3

Hauptwissensgebiete

- Verständnis von Source Code Repositories (öffentlich und privat)
- Verständnis der Grundsätze des Source Code Managements und der Organisation von Repositories
- Kenntnis der gängigen SCM-Systeme (Git, Subversion, CVS)
- Kenntnis der Begriffe Versionskontrollsystem (VCS), Revisionskontrollsystem und Source Code Managementsystem (SCM)

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Source Code Repositories
- Commits, Branches und Tags
- Feature-, Entwicklungs- und Release-Branches
- Subrepositories
- Code-Merges

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	056 Zusammenarbeit und Kommunikation
Lernziel:	056.2 Source Code Management
Lektion:	1 von 1

Einführung

Jeder, der schon einmal ein Textdokument im Team erstellt hat, kennt die Probleme einer solchen Zusammenarbeit: Welche Version ist die aktuelle? Wo ist diese Version gespeichert? Wird sie gerade von jemandem bearbeitet? Wer hat wann und warum welche Kommentare hinzugefügt oder Änderungen vorgenommen? Das Ergebnis sind oft unterschiedliche Versionen des Dokuments und im schlimmsten Fall eine Sammlung von Versionen, die niemand mehr überblickt.

Stellen Sie sich nun ein Softwareprojekt mit Hunderten von Dateien vor, an dem Entwickler aus der ganzen Welt arbeiten, indem sie neue Funktionen entwickeln, Fehler beheben, Teile abspalten und separat entwickeln usw. Ein solcher Entwicklungsprozess ist ohne geeignete Werkzeuge nicht mehr beherrschbar.

Spezielle Software für das *Source Code Management* (SCM), auch bekannt als *Version Control Systems* (Versionskontrollsysteme, VCS) oder *Revision Control Systems* (RCS), schafft hier Abhilfe und beseitigt die soeben skizzierten Probleme.

In der Welt der Softwareentwicklung ist SCM eine fundamentale Säule, die die Integrität der

Codebasis schützt. Stellen Sie es sich als einen gewissenhaften Wächter vor, der jede Änderung am Quellcode im Laufe der Zeit akribisch verfolgt.

Source Code Management System und Repository

Das Source Code Management System ist das Herzstück eines Softwareprojekts. Obwohl wichtige Arbeiten in Diskussionsforen und an anderen Stellen stattfinden, repräsentiert das SCM-System sowohl die Geschichte des Projekts als auch seinen aktuellen Stand als dynamische Einheit.

Das *Repository* des Quellcodes ist eine Art digitale Werkstatt für ein Projekt: Wie in einer physischen Werkstatt alle Werkzeuge und Materialien aufbewahrt werden, die zur Herstellung eines Werkstücks benötigt werden, so sind in einem Source Code Repository alle Dateien, Dokumente und der Code des Softwareprojekts gespeichert. Es bietet eine strukturierte Umgebung für Organisation und Verwaltung der Projektressourcen.

Informationen werden in der Regel in Form eines Verzeichnisbaums gespeichert. SCM-Systeme setzen meist auf ein Client-Server-Modell, bei dem jeder Benutzer Daten aus dem Repository abrufen und in das Repository einpflegen kann. Das System zeichnet auf, wer wann welche Änderungen vorgenommen hat, und sorgt so für Transparenz und Verantwortlichkeiten innerhalb des Teams.

Stellen Sie sich vor, Sie arbeiten an einem Projekt mit mehreren Entwicklern und es wird ein Fehler im Code entdeckt. Mit einem SCM können Sie ganz einfach die Änderungen zwischen den Versionen untersuchen und die spezifische Änderung am Code ermitteln, die zu dem Fehler geführt hat.

Da sich das System jede Version einer Datei merkt, sobald sie sich ändert, hat ein Benutzer Zugriff auf jede dieser Versionen und kann zu früheren Versionen zurückkehren (falls beispielsweise falsche Änderungen vorgenommen wurden). Das Repository ist also auch eine Art Archiv, das Zugriff auf jede Änderung, die jemals an dem Projekt vorgenommen wurde, und auf den Status des Projekts zu jedem Zeitpunkt in seiner Geschichte gewährt.

SCM-Systeme sparen Platz, indem sie die Änderungen an einer Datei nachverfolgen, anstatt bei jeder Änderung die gesamte Datei zu speichern. Diese effiziente Speichermethode sorgt dafür, dass frühere Versionen zugänglich sind, ohne übermäßig viele Ressourcen zu verbrauchen.

Viele Tools, wie etwa Continuous Integration/Continuous Delivery (CI/CD) und Testing, sind eng mit dem SCM-System verbunden. Außerdem bauen die Mitwirkenden ihre Reputation über das System auf, indem sie ihre Beiträge für alle sichtbar machen. Bei der Verwaltung des Quellcodes geht es also nicht nur darum, Änderungen zu verfolgen, sondern auch darum, die Integrität der Codebasis zu bewahren und die Zusammenarbeit zwischen den Entwicklern zu fördern; so ist

sichergestellt, dass sich Projekte dynamisch unter sich ständig verändernden Bedingungen weiterentwickeln können.

Beliebte SCM-Systeme sind Git, Subversion und CVS. Wie viele andere Softwarefunktionen wird SCM heute oft von spezialisierten Anbietern bereitgestellt. Mit anderen Worten, es handelt sich um Software as a Service (SaaS) oder einen "Cloud"-Dienst: Die Mitwirkenden haben die Software auf ihren lokalen Systemen, um ihre persönlichen Änderungen zu verwalten, laden ihre Änderungen aber in ein zentrales Repository hoch, das von typischen Cloud-Funktionen wie 24/7-Betriebszeit, Backups und sicherem Zugriff profitiert.

Millionen von Entwicklern nutzen inzwischen Cloud-Dienste, insbesondere GitHub. GitLab ist eine Alternative, die auf Open Source Code basiert. Sowohl GitHub als auch GitLab ermöglichen es, in den Cloud-Repositories des Anbieters zu arbeiten oder eine lokale Installation des SCM-Systems einzurichten. Ein Vorteil der Arbeit in diesen Systemen ist die Reputation, die man durch "Sterne" erwerben kann, indem andere Benutzer die eigene Arbeit bewerten. Cloud-Dienste bieten weitere Extras, wie Tracker für Änderungsanforderungen, Bewertungssysteme, Wikis und Diskussionsforen.

Repositories können sowohl persönliche als auch Unternehmensprojekte enthalten, sie sind also keineswegs auf ein kommerzielles Umfeld beschränkt. Jeder Entwickler, der ein Projekt starten oder an einem Open-Source-Projekt arbeiten und es nach seinen Bedürfnissen kopieren und verändern möchte, kann dies tun. In all diesen Fällen ist es wichtig, genau zu kontrollieren, wer auf das Repository zugreifen darf.

Um sich in der Versionskontrolle und dem Source Code Management zurechtzufinden, ist es wichtig, die Begrifflichkeiten zu kennen. In den folgenden Abschnitten werden wir darum einige dieser Konzepte und Begriffe klären.

Commits, Tags und Branches

Ein Entwickler erstellt jedes Mal einen *Commit*, wenn er eine Änderung in das Respository überträgt. Commits stellen Momentaufnahmen von Änderungen dar, die zu einem bestimmten Zeitpunkt an der Codebasis vorgenommen wurden. Jeder Commit enthält Metadaten wie den Namen des Autors, einen Zeitstempel und eine beschreibende Nachricht, die die Änderungen erläutert. Commits helfen Entwicklern, die Entwicklung der Codebasis nachzuverfolgen und die Geschichte bestimmter Änderungen zu verstehen.

Stellen Sie sich ein Team von Entwicklern vor, die an einer Webanwendung arbeiten. Jedes Mal, wenn sie Änderungen an der Codebasis vornehmen, erstellen sie einen Commit, um diese Änderungen zu dokumentieren. Zum Beispiel könnte jemand, der eine neue Funktion zur Anwendung hinzufügt, einen Commit mit einer Nachricht wie "Funktion zur

Benutzerauthentifizierung hinzugefügt" erstellen. Dieser Commit erfasst den Zustand der Codebasis, nachdem die Funktion implementiert wurde.

Behebt ein Entwickler einen Fehler, bezieht sich die Commit-Nachricht normalerweise auf die Nummer des Fehlers im Bug Tracking System, der Fehlerdatenbank, des Projekts.

Tags sind benannte Verweise auf bestimmte Commits. Sie markieren typischerweise wichtige Punkte in der Projektgeschichte, wie beispielsweise Releases oder Milestones. Tags bieten eine Möglichkeit, wichtige Versionen der Codebasis zu kennzeichnen und darauf zu verweisen, was die Verwaltung und Navigation in der Projektgeschichte erleichtert.

Branches, also Zweige oder Abzweigungen, kommen ins Spiel, wenn Entwickler gleichzeitig an sich überschneidenden Aufgaben arbeiten müssen. Branches sind unabhängige Entwicklungslinien, die von der Hauptcodebasis abweichen. So können Entwickler isoliert an Funktionen oder Korrekturen arbeiten, ohne den Hauptcode zu beeinflussen, bis die Änderungen für die Integration bereit sind. Branches helfen, die Entwicklungsarbeit zu organisieren, und erleichtern die Zusammenarbeit zwischen den Teammitgliedern.

Wenn zum Beispiel ein Entwickler an einer neuen Funktion für die Anwendung arbeitet, während ein anderer einen Fehler behebt, kann jeder von ihnen einen eigenen Branch erstellen, um die Änderungen zu isolieren. Sobald ihre Arbeit abgeschlossen ist, können sie ihre Zweige wieder in der Hauptcodebasis zusammenführen, was als *Merge* bezeichnet wird (Branches).

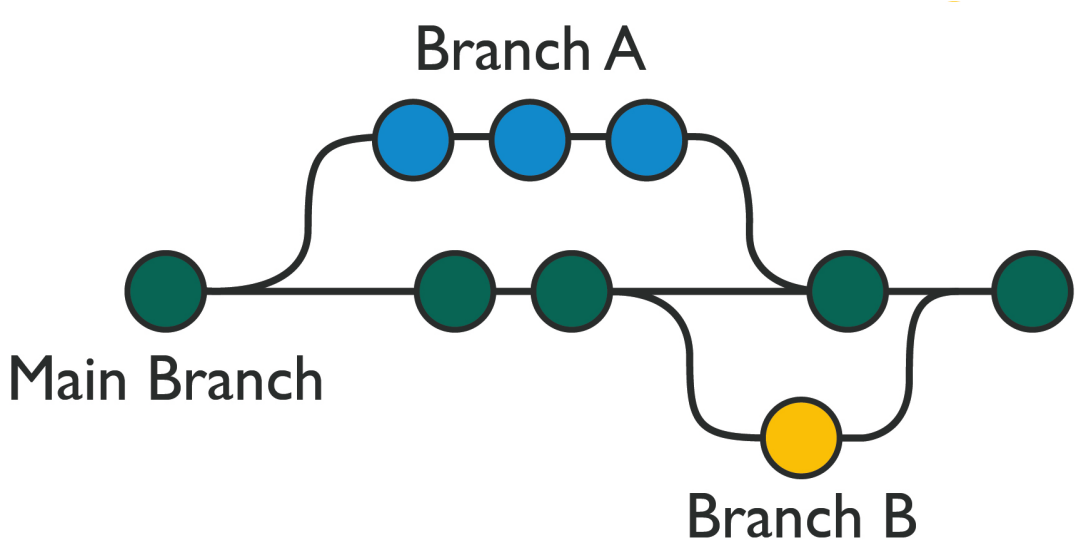

Figure 16. Branches

Wenn mehrere Personen an der gleichen Datei arbeiten oder Dateien in einen anderen Branch überführen, kann es vorkommen, dass sie Änderungen an derselben Zeile einer Datei vornehmen. Wenn die zweite Person ihre Änderungen wieder zurück in das Projekt zu übertragen versucht,

meldet das System einen *Konflikt*.

Einige Versionskontrollsysteme lassen es nicht zu, dass ein Entwickler eine Datei eincheckt, solange sie in Konflikt mit der aktuellen Version im Repository steht. Der Entwickler muss dann die aktuelle Version auschecken, den Konflikt lösen und eine neue Version einchecken.

Cloud-basierte Systeme bieten *Merge* oder *Pull Requests*. Diese werden von einem Entwickler erstellt, der in einem lokalen System oder Branch gearbeitet hat und glaubt, dass seine Änderungen für die Aufnahme in einen anderen Branch geeignet sind. Die Entwickler des Zielzweigs entscheiden dann, ob sie die Anfrage annehmen.

Subrespositories

Oft wird für die Entwicklung eines Softwareprojekts (beispielsweise eine komplexe Website) der Code eines anderen, unabhängigen Projekts (etwa ein Mediaplayer) benötigt. Statt den Code des Mediaplayers ganz oder teilweise in das eigene Projekt zu kopieren, bieten viele VCS die Möglichkeit von *Subrepositories*, auch *Submodule* genannt.

In unserem Beispiel wird das Repository des Mediaplayers als Subrepository in das Repository der Website integriert und erscheint dann als separates Verzeichnis im Verzeichnisbaum. Das bedeutet, dass die Codebasis des Mediaplayers vollständig verfügbar ist, aber unabhängig bleibt und bei Bedarf sogar aus dem ursprünglichen Repository des Mediaplayers aktualisiert werden kann.

Dieses Feature erweist sich bei der Organisation komplexer Projekte mit zahlreichen Abhängigkeiten oder der Integration von Bibliotheken und Frameworks von Drittanbietern in eine Codebasis als überaus hilfreich. Submodule verbessern die Projektorganisation und erleichtern die Zusammenarbeit, indem sie Entwicklern die Möglichkeit geben, effizient mit dem Code verschiedener (Teil-)Projekte zu arbeiten.

Verwendung eines Source Code Management Systems

Jeder Mitwirkende an einem Projekt beginnt mit der Erstellung einer Identität, die in der Regel mit seiner eindeutigen E-Mail-Adresse verknüpft ist. Cloud-basierte Systeme verwalten Identitäten über Konten, wie es auch Social Media Sites und andere Organisationen tun.

Neue Entwickler durchlaufen eine Prozedur wie die folgende, wenn sie ein SCM-System verwenden:

1. SCM-Software installieren, wenn sie nicht bereits auf dem eigenen System vorhanden ist.
2. Das gesamte Projekt einmalig auf das lokale System übertragen — ein Vorgang, der als *Klonen*

bezeichnet wird.

3. Ab diesem Schritt lokal arbeiten (ggf. in einem oder mehreren Branches) und Änderungen in das Repository übertragen (*Push*).
4. Vor jeder neuen Arbeitssitzung die aktuelle Version aus dem Repository holen (*Pull*).

Die Projektleiter entscheiden, wem sie vertrauen und Zugang zum Repository gewähren. Senior Developer haben die wichtige Aufgabe zu entscheiden, wann die von anderen Mitwirkenden eingereichten Änderungen in den Hauptzweig des Repositorys aufgenommen werden.

In Cloud-basierten Systemen kann der *Owner* eines Projekts den Zugang zu einem Repository auch dadurch steuern, dass er dessen Sichtbarkeit auf “öffentlich” oder “privat” einstellt. Öffentliche Repositories gewähren jedermann Lesezugriff über das Internet. Private Repositories hingegen beschränken den Zugriff auf den Eigentümer, auf Personen, für die der Eigentümer den Zugriff ausdrücklich freigegeben hat, und im Falle von Organisations-Repositories auf bestimmte Mitglieder der Organisation.

Stellen Sie sich ein Team von Entwicklern vor, die an einer E-Commerce-Website arbeiten. Sie beschließen, eine neue Funktion hinzuzufügen, über die Kunden ihre Bestellungen nachverfolgen können. Um diese Funktion zu implementieren, erstellen sie einen *Feature Branch* mit einem Namen wie `order-tracking`; darin nehmen sie die notwendigen Änderungen im Code vor, ohne die Hauptcodebasis zu ändern. Sobald die Funktion fertiggestellt und getestet ist, fügen sie diesen Branch in den Hauptentwicklungszweig (*Development Branch*) für weitere Integration und Tests ein.

Der Hauptentwicklungszweig dient als zentraler Knotenpunkt, in dem alle neuen Funktionen zum Testen zusammengeführt werden. Wenn beispielsweise mehrere Entwickler gleichzeitig an verschiedenen Funktionen arbeiten, können sie ihre Feature Branches mit dem Development Branch zusammenführen, um sicherzustellen, dass alles reibungslos funktioniert. Dieser Integrationsprozess hilft dabei, etwaige Konflikte oder Kompatibilitätsprobleme frühzeitig zu erkennen und zu beheben.

Wenn es an der Zeit ist, eine neue Version der E-Commerce-Website freizugeben, erstellt das Team aus dem Development Branch einen *Release Branch*, beispielsweise `v2.0`. Sie konzentrieren sich darauf, die Codebasis zu stabilisieren, letzte Fehler zu beheben und gründliche Tests durchzuführen, um eine reibungslose Veröffentlichung zu gewährleisten. Ist die Freigabe erteilt, wird der Code aus dem Release Branch in die produktiven Systeme ausgeliefert (*Deploy*), und der Zyklus beginnt von neuem.

Bekannte Versionskontrollsysteme

Einige der bekanntesten Versionskontrollsysteme sind Git, Subversion (auch bekannt als SVN) und CVS. All diese Systeme sind Open Source.

Git ist ein verteiltes Versionskontrollsystem, das in der Softwareentwicklung und anderen Bereichen weit verbreitet ist. Bei der Arbeit mit Git hat jeder Entwickler eine vollständige Kopie der Codebasis auf seinem Computer.

Der dezentrale Ansatz ermöglicht es den Entwicklern, offline zu arbeiten und nahtlos zusammenzuarbeiten, ohne auf einen zentralen Server angewiesen zu sein. Das heißt, sie können unabhängig voneinander an verschiedenen Funktionen oder Fehlerbehebungen arbeiten sowie Änderungen nahtlos zusammenführen oder Updates untereinander austauschen, selbst wenn der zentrale Server offline geht.

Erinnern Sie sich an die Entwicklung des Linux-Kernels, der ursprünglich auf ein zentralisiertes Versionskontrollsystem namens BitKeeper angewiesen war. Als der kostenlose Status von BitKeeper aufgehoben wurde, entwickelten Linus Torvalds und die Linux Community Git als verteilte Alternative. Diese Entscheidung machte eine nicht-lineare und hoch effiziente Entwicklung großer Projekte wie des Linux-Kernels erst möglich. Der Erfolg von Git für den Linux-Kernel — ein extrem komplexes Projekt mit Tausenden von Entwicklern und unzähligen Branches — zeigt die Leistungsfähigkeit und Skalierbarkeit des Systems.

Die meisten Softwareentwickler setzen heute auf Git, und auch beliebte SaaS-Angebote bauen darauf auf.

Git behandelt Konflikte, indem es dem Entwickler eine Datei mit beiden Versionen der geänderten Zeile zeigt und deutlich markiert, aus welcher Version der Datei die Zeilen stammen. Der Entwickler muss entscheiden, wie er den Konflikt auflöst und eine kohärente Version der Datei eincheckt.

Subversion (SVN) war vor Git das wahrscheinlich populärste SCM-System. Im Gegensatz zu Git ist Subversion zentralisiert: Die Versionsgeschichte liegt auf einem zentralen Server. Die Entwickler verbinden sich mit diesem Server, um Änderungen vorzunehmen, wodurch sichergestellt wird, dass jeder mit der neuesten Version der Codebasis arbeitet.

Angenommen Sie sind Teil eines Teams, das an einem Projekt mit Subversion arbeitet. Jedes Mal, wenn Sie Änderungen an der Codebasis vornehmen müssen, stellen Sie eine Verbindung zum zentralen SVN-Server her, um eine Arbeitskopie des Codes auszuchecken. So stellen Sie sicher, dass Sie mit der aktuellsten Version des Projekts arbeiten. Nachdem Sie Ihre Änderungen vorgenommen haben, übertragen Sie sie zurück auf den Server und aktualisieren das zentrale Repository mit Ihren Änderungen. Dieser zentralisierte Arbeitsablauf trägt dazu bei, die

Konsistenz des Projekts zu wahren.

Vor Subversion war *CVS* ein sehr beliebtes, zentralisiertes Versionskontrollsystem, das aber Designprobleme hatte, die zur Entwicklung von Subversion als Alternative führten.

Geführte Übungen

1. Nennen Sie drei Hauptmerkmale von SCM-Systemen.

2. Beschreiben Sie das Konzept des Tagging in SCM-Systemen und erklären Sie, warum es für die Verwaltung von Software Releases wichtig ist.

3. Was ist der Unterschied zwischen einem Branch und einem Subrepository in einem SCM-System?

Offene Übungen

1. Vergleichen Sie Git und Subversion (SVN) in Bezug auf Architektur und Arbeitsabläufe.

2. Was ist der “Index” oder die “Staging Area” in Git?

3. Skizzieren Sie die Trunk-basierte Verzweigungsstrategie von Git.

4. Welche der folgenden SCM-Systeme sind Open Source?

Git	
Mercurial	
Subversion	
GitHub	
Bitbucket	
GitLab	

Zusammenfassung

In dieser Lektion ging es um die zentrale Rolle von Source Code Management Systemen in der modernen Softwareentwicklung. Sie haben die grundlegenden Begriffe und Möglichkeiten der Nutzung solcher Systeme kennengelernt, einschließlich Repository, Branches, Tags und Merges.

Antworten zu den geführten Übungen

1. Nennen Sie drei Hauptmerkmale von SCM-Systemen.
   - Änderungen am Quellcode protokollieren
   - (Gleichzeitigen) Zugriff der Entwickler auf den Quellcode verwalten
   - Beliebigen Entwicklungsstand von Dateien oder des gesamten Projekts wiederherstellen
2. Beschreiben Sie das Konzept des Tagging in SCM-Systemen und erklären Sie, warum es für die Verwaltung von Software Releases wichtig ist.

   Unter Tagging versteht man das Zuweisen von beschreibenden Bezeichnungen oder Namen zu bestimmten Commits innerhalb der Codebasis, um wichtige Punkte in der Projektgeschichte zu markieren, beispielsweise Releases oder Milestones. Diese Tags bieten eine bequeme Möglichkeit, auf bestimmte Versionen der Codebasis zu verweisen und die Entwicklung des Projekts im Laufe der Zeit nachzuverfolgen.

3. Was ist der Unterschied zwischen einem Branch und einem Subrepository in einem SCM-System?

   Ein Branch ist eine parallele Entwicklungslinie in einem Projekt, etwa zur Fehlerbehebung oder zur Entwicklung neuer Funktionen, die in der Regel wieder in den Hauptentwicklungszweig eingebunden wird, sobald die Aufgabe abgeschlossen ist. Ein Subrepository oder Sumbmodul ist ein unabhängiges Projekt, dessen Repository in ein Projekt integriert wird, um auf dessen Codebasis zugreifen zu können. Das Subrepository erscheint als Verzeichnis im Verzeichnisbaum des Projekts und bleibt unabhängig.

Antworten zu den offenen Übungen

1. Vergleichen Sie Git und Subversion (SVN) in Bezug auf Architektur und Arbeitsabläufe.

   Git ist ein verteiltes Versionskontrollsystem (Distributed Version Control System, DVCS), das jedem Entwickler eine vollständige Kopie der Codebasis überlässt und ihm ermöglicht, sogar offline am Quellcode zu arbeiten. Git ist aufgrund seiner Geschwindigkeit, Flexibilität und robusten Branch- und Merge-Funktionen sehr beliebt. SVN ist ein zentralisiertes VCS, das den Versionsverlauf auf einem zentralen Server speichert. In bestimmten Unternehmensumgebungen ist es aufgrund der zentralisierten Struktur und seines ausgereiften Funktionsumfangs nach wie vor beliebt.

2. Was ist der “Index” oder die “Staging Area” in Git?

   Der Index oder Staging-Bereich ist eine Zwischenschicht zwischen der lokalen Arbeitskopie des Projekts und der aktuellen Version auf dem Server. Es handelt sich um eine Datei, in der alle Informationen für den nächsten Commit eines Benutzers gespeichert sind.

3. Skizzieren Sie die Trunk-basierte Verzweigungsstrategie von Git.

   Die Trunk-basierte Entwicklung ist eine Strategie, bei der die häufige Integration von Änderungen in die Hauptcodebasis (den *Trunk*) im Vordergrund steht. Entwickler arbeiten an kurzlebigen Feature Branches und führen sie gegebenenfalls mehrmals am Tag in den Trunk ein, um kontinuierliche Integration und schnelles Feedback zu gewährleisten.

4. Welche der folgenden SCM-Systeme sind Open Source?

Git	X
Mercurial	X
Subversion	X
GitHub	
Bitbucket	
GitLab	X

056.3 Werkzeuge für Zusammenarbeit und Kommunikation

Referenz zu den LPI-Lernzielen

Open Source Essentials version 1.0, Exam 050, Objective 056.3

Gewichtung

2

Hauptwissensgebiete

- Verständnis gängiger Kommunikationswerkzeuge
- Verständnis der gängigen Methoden zur Erfassung und Sicherung von Wissen
- Verständnis gängiger Werkzeuge für Informationsmanagement und Veröffentlichung
- Verständnis gängiger Arten von Dokumentation
- Verständnis der gängigen Funktionen der Zusammenarbeit von Plattformen des Source Code Management

Auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme

- Instant Messenger
- Chat-Plattformen
- Mailinglisten
- Newsletter
- Issue Tracker und Bug Tracker
- Bug Reports
- Merge-Anfragen und Pull-Anfragen
- Helpdesk- und Ticketingsysteme
- Wikis

- Dokumentenverwaltungssysteme (DMS)
- Dokumentationswebsites
- Produktwebsites
- Content Management Systeme (CMS)
- Architektur-Dokumentation
- Benutzerdokumentation
- Administratordokumentation
- Entwicklerdokumentation

Linux Professional Institute

Lektion 1

Zertifikat:	Open Source Essentials
Version:	1.0
Thema:	056 Zusammenarbeit und Kommunikation
Lernziel:	056.3 Werkzeuge für Zusammenarbeit und Kommunikation
Lektion:	1 von 1

Einführung

Viele Open-Source-Projekte haben aktive Teilnehmer auf der ganzen Welt. Die Zusammenarbeit findet meist “virtuell” statt, und die Beteiligten sind oft über Länder, Kontinente und Zeitzonen verteilt. Außerdem haben sie in der Regel verschiedene Muttersprachen. Kurz gesagt: Egal wo Sie sich auf der Welt befinden oder welche Sprache Sie sprechen, Sie können zu einem Open-Source-Projekt beitragen!

Diese Vielfalt macht die Mitwirkung an einem Open-Source-Projekt überaus interessant, da man seinen Horizont erweitern und viel lernen kann; gleichzeitig können die Herausforderungen in Sachen Kommunikation und Koordination beträchtlich sein. Gute und schnelle Kommunikation ist der Schlüssel zum Erfolg und zur Nachhaltigkeit eines jeden Open-Source-Projekts, und um diese zu gewährleisten, haben Open-Source-Projekte entsprechende Strukturen geschaffen und nutzen eine Vielzahl von Werkzeugen, die die Zusammenarbeit effizienter machen.

Eine weitere Herausforderung besteht darin, dass Open-Source-Projekte, die oft von Freiwilligen betrieben werden, mit einer gewissen Fluktuation in der Beteiligung konfrontiert sind. Das Leben und die Hobbys der Beteiligten verändern sich — manche verlieren vielleicht das Interesse oder

haben einfach keine Zeit mehr. Vielleicht tragen sie jahrelang zu einem Open-Source-Projekt bei, aber wenn sie den Job wechseln, heiraten oder Kinder großziehen, reicht einfach nicht mehr die Zeit für ehrenamtliches Engagement.

Daher sollten Open-Source-Projekte neben der Bereitstellung effizienter Kommunikationsmittel auch die Dokumentation und Weitergabe von Wissen sicherstellen, damit das Rad nicht immer wieder neu erfunden werden muss. Die Abrufbarkeit von Informationen hilft den Mitwirkenden, aus den vergangenen Fehlern anderer zu lernen und dieselben Fehler zu vermeiden.

In dieser Lektion geht es um gängige Werkzeuge der Zusammenarbeit in einem Open-Source-Projekt und wie Sie in einer internationalen Community kommunizieren. Es gibt für jeden einen einfachen Weg, seinen ersten Beitrag zu leisten!

Als Beispiel für Kommunikation und Informationsaustausch sehen wir uns *LibreOffice* an, eine Open Source Office Suite, die in mehr als hundert Sprachen verfügbar ist. Die Benutzer sind gelegentliche Privatanwender ebenso wie große Verwaltungen und Unternehmen. Entsprechend groß ist die Zahl der Mitwirkenden — nicht nur in der Entwicklung, sondern auch bei Lokalisierung, Dokumentation, Marketing, Qualitätssicherung, Infrastruktur, Grafik- und UX-Design und vieles mehr.

Mit anderen Worten: Bei LibreOffice, wie auch bei vielen anderen Open-Source-Projekten, können Sie Ihre Fähigkeiten und Talente in dem Bereich einbringen, in dem Sie sich wohlfühlen. Sie müssen nicht unbedingt technisch geschult oder gar professioneller Entwickler sein — ebenso gut können Sie Ihre Kreativität, Ihr künstlerisches Talent oder Ihre Sprachkenntnisse einbringen. Das Projekt hat eine Website, die die verschiedenen Bereiche für Beiträge dargestellt: https://whatcanidoforlibreoffice.org. LibreOffice ist daher ein gutes Besipiel für die vielen Aspekte der Zusammenarbeit in einer Community.

Kommunikationswege

Bevor es um Details der Kommunikationsmittel geht, ist es wichtig zu verstehen, wie Kommunikation im Allgemeinen funktioniert, da diese Überlegungen die Wahl der Mittel bestimmen.

In Open-Source-Projekten gibt es zwei Arten der Kommunikation: Bei der *synchronen* Kommunikation tauschen sich Menschen gleichzeitig aus. Beispiele sind natürlich persönliche Gespräche, aber auch Videokonferenzen oder Telefonate. Bei der *asynchronen* Kommunikation werden Nachrichten zeitlich versetzt ausgetauscht. Beispiele sind E-Mail und SMS, aber auch ein Postbrief oder ein Telefax.

Diese Unterscheidung ist jedoch nicht immer leicht zu treffen: Eine Messenger-Anwendung auf

einem Mobiltelefon (WhatsApp, Telegram, Signal oder Element) ist technisch gesehen eine asynchrone Kommunikation. Wenn jedoch beide Gesprächspartner zur gleichen Zeit online sind und sofort antworten, gehen sie ein direktes Gespräch ein.

Das Beispiel zeigt, dass ein Teil der Kommunikation auch davon abhängt, wie die Mitwirkenden ihre Werkzeuge nutzen und welche Erwartungen sie haben. Jede Aufgabe kann einen anderen Satz von Kommunikationswerkzeugen erfordern. Die folgenden Abschnitte befassen sich damit im Detail.

Synchrone Kommunikation

Synchrone Kommunikation ist ebenso effektiv wie anspruchsvoll und findet zu einem bestimmten Zeitpunkt im gleichen physischen oder virtuellen Raum statt.

Ein direktes Treffen ist ideal, um Fragen interaktiv zu klären, anstatt lange E-Mail-Nachrichten auszutauschen, die das Risiko von Missverständnissen mit sich bringen. Stellen Sie sich vor, Sie wollen mehr über ein Open-Source-Projekt erfahren und die Mitglieder der Community kennenlernen; E-Mail-Nachrichten und Websites können bei der Einführung helfen und haben niedrigere Einstiegshürden, aber ein nachhaltiger erster Eindruck entsteht, wenn Sie mit jemandem persönlich sprechen. Es sind nicht zuletzt diese direkten Interaktionen, die an einem Open-Source-Projekt faszinieren und dazu motivieren, selbst etwas beizutragen.

Neben dem gegenseitigen Kennenlernen eignen sich synchrone Treffen insbesondere auch bei Konflikten oder Problemen — oder wenn man negative Nachrichten überbringen muss.

Aber natürlich sind internationale Projekte mit einer Herausforderung konfrontiert, die keine Technologie überwinden kann: Zeitzonen. Bei Mitwirkenden auf verschiedenen Kontinenten ist es schwierig, Zeitfenster für Besprechungen zu finden, die allen passen. In Australien wacht vielleicht gerade jemand auf, wenn jemand in Europa kurz davor ist, Feierabend zu machen. Eine weitere Herausforderung besteht darin, dass manche Beiträger nur nachts oder am Wochenende Zeit finden, während andere die Bürozeiten an Werktagen bevorzugen.

Hinzu kommt, dass nicht jeder fließend Englisch spricht — und noch gibt es keine allgemein zugänglichen und verlässlichen Werkzeuge zum Simultandolmetschen.

Dennoch sind Videokonferenzen eines der häufigsten Kommunikationsmittel in Open-Source-Projekten. Das LibreOffice-Projekt führt regelmäßig Online Meetings für seine Community durch, beispielsweise für Entwickliung, Marketing, Infrastruktur, Qualitätssicherung, Benutzererfahrung und Design.

Asynchrone Kommunikation

Über asynchrone Kommunikation beteiligt man sich an einer Diskussion zu einem Zeitpunkt und in einer Geschwindigkeit, die einem selbst angemessen scheint. Bekanntestes Beispiel ist wahrscheinlich E-Mail: Sie antworten auf eine Nachricht, wann immer Sie es für angemessen halten — sei es in der nächsten Minute oder am nächsten Tag.

Asynchrone Kommunikation erfolgt meist schriftlich, was auch eine maschinelle Übersetzung erlaubt. So können Sie Nachrichten in einer Fremdsprache lesen und verstehen und sogar Ihre Antwort rückübersetzen.

Niedergeschriebene Inhalte sind auch leichter zu behalten oder in Dokumentation zu überführen. Einem Benutzer am Telefon die Funktion einer Software zu erklären ist deutlich schwieriger, als die Erklärungen in einem Support-Dokument aufzubereiten.

Interne und externe Kommunikation

Nicht zuletzt hängt Kommunikation auch davon ab, ob es sich um interne oder externe Empfänger handelt. Sie werden eine interne Notiz an die Systemadministratoren anders verfassen als eine Pressemitteilung für Hunderte Journalisten. Bedenken Sie jedoch, dass aufgrund der Natur eines Open-Source-Projekts Kommunikation, die ursprünglich vielleicht nicht explizit für die Öffentlichkeit bestimmt war, öffentlich werden kann, beispielsweise in den Archiven von Mailinglisten (im Fall von LibreOffice ist es https://listarchives.documentfoundation.org/).

Kommunikationswerkzeuge

Vor dem Hintergrund der genannten allgemeinen Aspekte von Kommunikation geht es in den folgenden Abschnitten um verschiedene Werkzeuge, die dazu in einem Open-Source-Projekt üblicherweise zum Einsatz kommen.

E-Mail, Mailinglisten und Newsletter

Eines der ersten Tools, das Sie in der Kommunikation mit einem Open-Source-Projekt nutzen werden, ist die “klassische” E-Mail. Viele Projekte betreiben *Mailinglisten*, die im Wesentlichen Verteilerlisten für E-Mails sind: Mit einer E-Mail-Nachricht erreichen Sie hunderte oder sogar tausende von Abonnenten, die an bestimmten Themen interessiert sind.

Mailinglisten gehören zu den ältesten Tools von Open-Source-Projekten und dienen sowohl der projektinternen Koordination als auch der Interaktion mit den Benutzern. Wenn Sie eine Frage zur Software haben oder einen Fehler im Programm melden wollen, ist die Wahrscheinlichkeit

groß, dass es dafür eine Mailingliste gibt. Die LibreOffice-Community beispielsweise bietet eine Vielzahl internationaler und lokaler Mailinglisten für verschiedene Themen an (https://www.libreoffice.org/get-help/mailing-lists/), von der Benutzerunterstützung über Diskussionen mit den Entwicklern bis hin zur Koordination der Infrastruktur (LibreOffice Webseite für Mailinglisten).

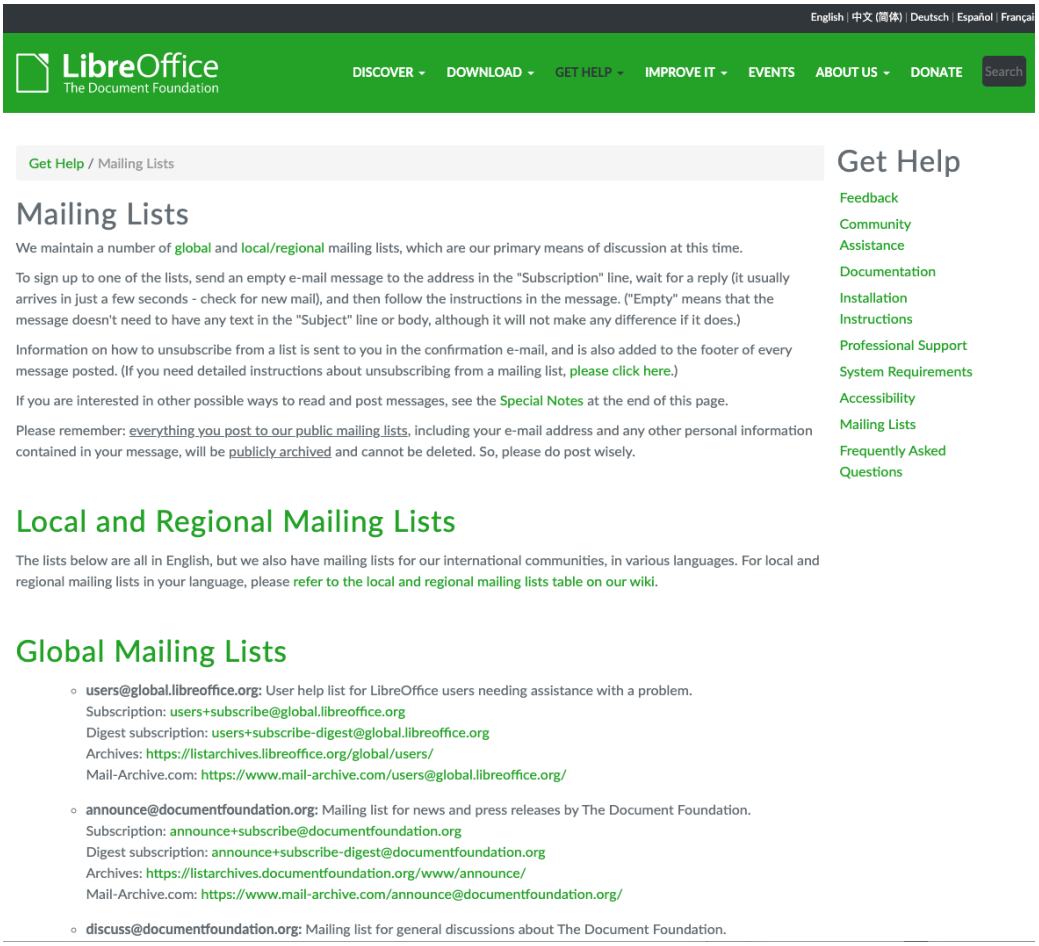

Figure 17. LibreOffice Webseite für Mailinglisten

Jede gesendete Nachricht wird in der Regel auch in einem öffentlichen *Mailinglisten-Archiv* gespeichert. Einmal versendet, kann eine Nachricht nicht mehr ohne weiteres gelöscht werden — gemäß der gängigen Redewendung: “Das Internet vergisst nichts.” Man sollte mit seinen Nachrichten darum durchaus vorsichtig sein. Vermeiden Sie beispielsweise die Angabe Ihrer privaten Adresse oder Telefonnummer in der Signatur oder den Versand vertraulicher Dokumente im Anhang.

Ein Nachteil von Mailinglisten ist, dass die Verwaltung in einem Mailprogramm nicht immer einfach ist. Sogenannte *Filter*, die auf bestimmten Elementen der Nachricht basieren (etwa einem Präfix in der Betreffzeile) helfen dabei. Die Feinheiten im Umgang mit einer großen Menge von E-Mails können für unerfahrene Benutzer schwierig sein. Daher gehen immer mehr Projekte zu

Diskussionsforen über — davon später mehr.

Eine besondere Form der Mailingliste ist der *Newsletter*. Wenn Sie über die neuesten Projektentwicklungen auf dem Laufenden bleiben und über neue Softwareversionen informiert werden möchten, abonnieren Sie den Newsletter und erhalten eine E-Mail-Nachricht, wenn etwas Wichtiges passiert.

Diskussionsforen

Abgesehen von dem Aufwand, den die Verwaltung von Mailinglisten in einem E-Mail-Programm erfordert, setzen vor allem jüngere Menschen immer weniger auf E-Mail als Kommunikationsmittel. Auch darum verlagern immer mehr Open-Source-Projekte ihre Kommunikation in *Diskussionsforen*. Die Grundidee ist mit E-Mail durchaus vergleichbar: Jedes Forum hat verschiedene Kategorien/Themen, in denen man sich austauscht, sogenannte *Threads*. Ähnlich wie bei E-Mail kann man in einem Forum mit dem Open-Source-Projekt in Kontakt treten und Aktivitäten koordinieren, Vorschläge für die Richtung des Projekts machen und als Benutzer Fehler melden.

Alles, was in einem Forum gepostet wird, ist in der Regel öffentlich sichtbar, genau wie in einer Mailingliste; aber anders als dort lassen sich Inhalte je nach Konfiguration des Forums auch nachträglich bearbeiten oder löschen. Die Benutzerfreundlichkeit von Foren ist, besonders für Einteiger, oft höher als die von Mailinglisten. Das LibreOffice-Projekt hat damit begonnen, einige seiner Mailinglisten in Foren umzuwandeln (https://community.documentfoundation.org/) und konnte seitdem eine steigende Beteiligung an den Diskussionen verzeichnen.

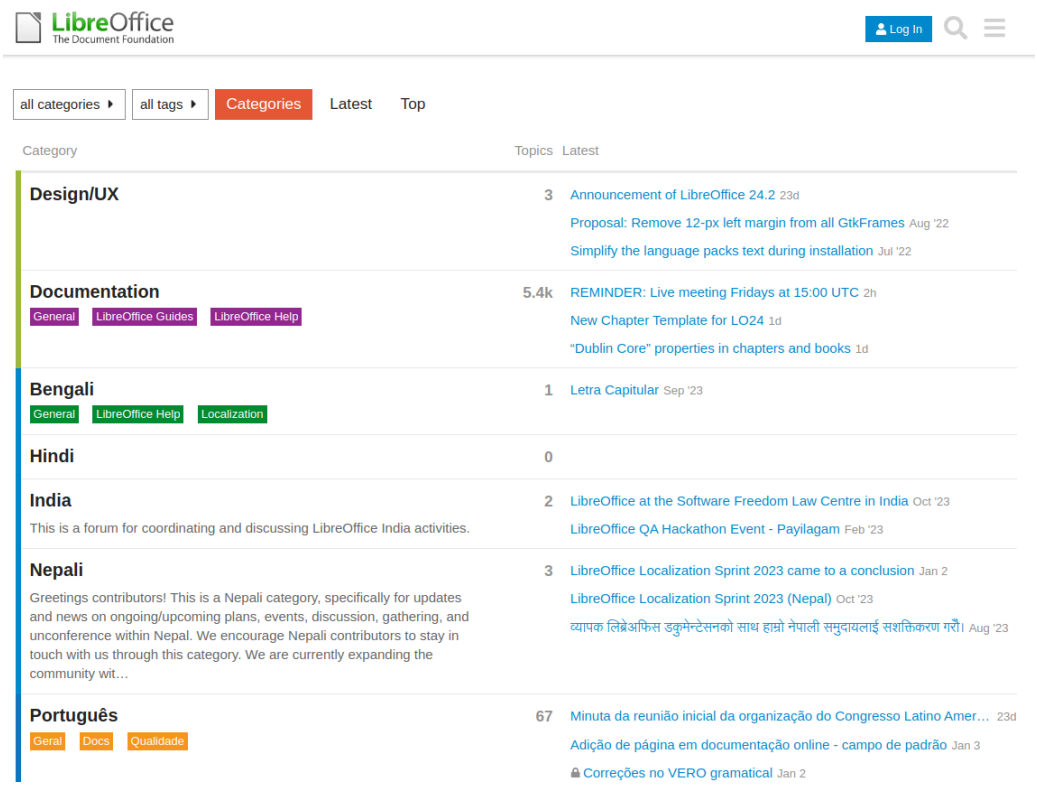

Figure 18. LibreOffice Webseite für Diskussionsforen

Instant Messages und Chat-Plattformen

Ein weiterer Weg, mit einer Open Source Community in Kontakt zu treten, sind Instant Messages (Sofortnachrichten) und Chat-Plattformen. Mit dem Aufkommen von Tools wie WhatsApp, Telegram, Signal oder Matrix hat fast jeder bereits eine dieser populären Anwendungen auf seinen Geräten installiert, was die Einstiegshürde deutlich niedriger macht. Sofortnachrichten sind auch bei jüngeren Leuten sehr viel beliebter als E-Mail oder Foren. Es überrascht daher nicht, dass viele Open-Source-Projekte diese heutzutage einsetzen.

Chat-Teilnehmer geben Nachrichten ein, die ähnlich wie E-Mails an alle anderen Teilnehmer gesendet werden. Je nach Chat-Plattform lässt sich eine Nachricht formatieren und mit Grafiken und Anhängen ergänzen.

Nachrichten-Apps sind ähnlich organisiert wie Foren oder E-Mail. Es stehen mehrere *Gruppen* oder *Channel* (Kanäle) zur Verfügung, an denen man sich je nach Interesse beteiligt. Nachrichten lassen sich in der Regel bearbeiten oder löschen; oft gibt es auch reine Ankündigungskanäle, deren Funktion der eines E-Mail-Newsletters entspricht.

Nachteil von Instant-Messenger-Apps ist, dass sie meist auf dem Mobiltelefon installiert sind, das auf jede neue Nachricht hinweist. Dies führt schnell zu einer Informationsflut bzw. "Alarmmüdigkeit". Mit der richtigen Konfiguration sind diese Benachrichtigungen jedoch zu

handhaben.

Ein weiterer Nachteil besteht darin, dass viele Messenger-Apps proprietär und in den Händen eines einzigen Anbieters sind, was die langfristige Bewahrung von Wissen erschwert, wenn die Daten nicht frei zugänglich sind.

Stand-alone, föderale und zentralisierte Kommunikation

Open-Source-Projekte arbeiten offen, auf der Grundlage offener Standards und offener Werkzeuge. Daher ist es wichtig zu verstehen, wie diese Werkzeuge im Hinblick auf Interoperabilität konzipiert sind. Drei Kategorien sind dabei zu unterscheiden.

Eine *stand-alone* Plattform läuft isoliert für eine Community. Beispiele sind Foren oder Wikis, die in der Regel nicht mit Instanzen anderer Projekte verbunden sind.

Dezentrale oder *föderale* Systeme laufen separat für jede Community, können aber miteinander verbunden werden. Ein Beispiel ist E-Mail: ein lokaler E-Mail-Server kann E-Mails an jeden anderen E-Mail-Server in der Welt senden. Andere Beispiele sind Nextcloud und ownCloud, die Dateifreigaben mit anderen Servern "föderieren", oder der Messenger-Dienst Element, über den man mit Benutzern anderer Server kommunizieren kann. Die gleichen Prinzipien gelten für das soziale Netzwerk Mastodon.

Sowohl stand-alone als auch verteilte Plattformen haben einen großen Vorteil: Das Open-Source-Projekt behält die volle Kontrolle über alle Inhalte und Funktionen. Das gesamte in einem solchen System gespeicherte Wissen bleibt in den Händen der Open Source Community und unterliegt nicht der Kontrolle durch Dritte.

Ein *zentralisiertes* System hingegen wird von einem Anbieter betrieben und interagiert nicht mit Dritten. Klassische Beispiele sind soziale Netzwerke wie Facebook oder Instagram oder Messenger-Apps wie WhatsApp oder Telegram. Alle Inhalte werden auf den Servern des externen Anbieters gespeichert und unterliegen dessen Geschäftsbedingungen.

Wenn Sie in einer Open Source Community aktiv sind, nutzen Sie wahrscheinlich alle drei Optionen. Zentralisierte Systeme eignen sich hervorragend, um viele Menschen zu erreichen, da sie oft beliebt sind und eine breite Benutzerbasis haben. Für die eigentliche Arbeit im Projekt ist jedoch ein föderales oder stand-alone System unter der Kontrolle der Community am besten geeignet.

Tools für die Zusammenarbeit

Die Unterscheidung zwischen Werkzeugen für die Kommunikation und solchen für die Zusammenarbeit ist nicht immer eindeutig. Allgemein machen Kommunikationswerkzeuge den

Austausch zwischen den Teilnehmern an einem Projekt überhaupt erst möglich, während Tools für die Zusammenarbeit die gemeinsame, konkrete Arbeit unterstützen.

Geeignete Tools für die Zusammenarbeit dienen etwa der Speicherung von Dateien, der Bearbeitung von Dokumenten in Echtzeit, der Nachverfolgung von Dokument- und Softwareversionen und vielem mehr. Während E-Mail oder Foren als Allzweckspeicher dienen, machen spezialisierte Tools das Wissen leichter zugänglich und die Zusammenarbeit effektiver.

Mit anderen Worten, es handelt sich um spezialisierte Werkzeuge für bestimmte Aufgaben. Wenn Sie zu einem Open-Source-Projekt beitragen wollen, werden Sie diese rasch kennenlernen.

Wikis

Eines der ältesten und beliebtesten Werkzeuge der Zusammenarbeit ist das *Wiki*. Berühmt geworden vor allem durch Wikipedia, ermöglicht ein Wiki den Benutzern die gemeinsame Arbeit an einer Website, die aus mehreren Dokumenten oder "Artikeln" besteht. Sie können in verschiedenen Kategorien gruppiert und nach Sprache gefiltert werden und enthalten Formatierungen, Tabellen und Bilder.

Wikis dienen oft als Wissensdatenbank, zu der jeder beitragen kann. Wenn Sie Content für ein Open-Source-Projekt liefern möchten, ist die Teilnahme an dessen Wiki ein guter Einstieg. Sie können bestehende Inhalte übersetzen, Artikel bearbeiten und aktualisieren oder neue Inhalte erstellen. Im Wiki des LibreOffice-Projekts (https://wiki.documentfoundation.org) finden Sie Marketingmaterial, Protokolle von Vorstandssitzungen, Installationsanleitungen und Konferenzplanung, und das alles in zahlreichen Sprachen (LibreOffice Wiki).

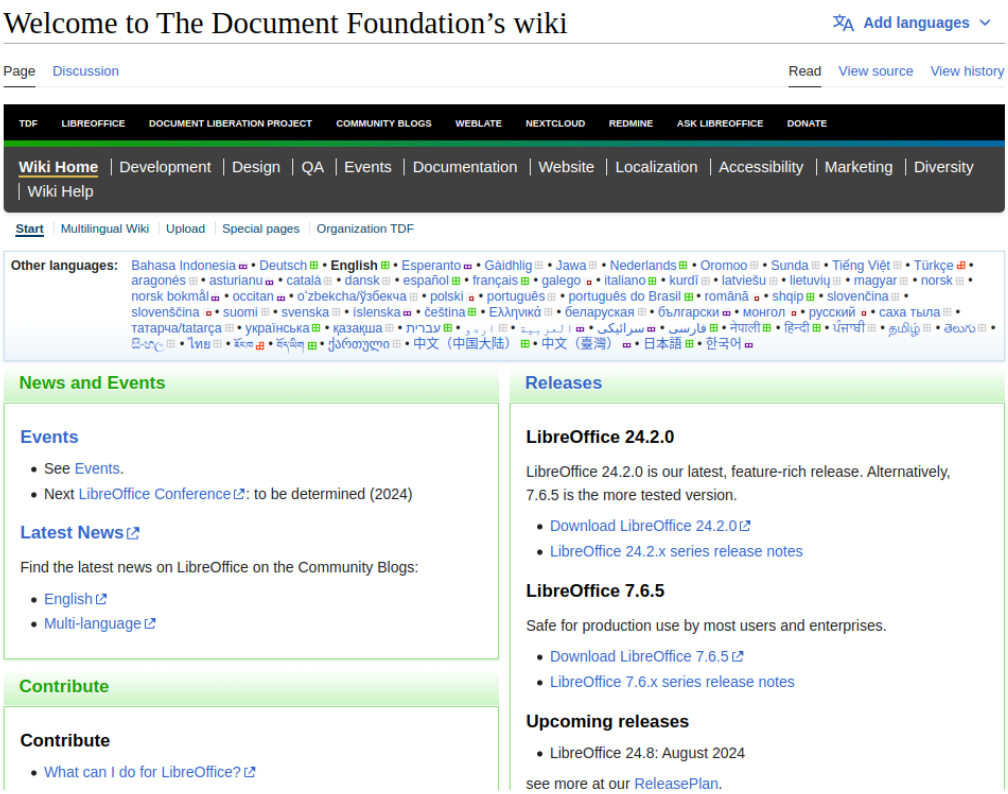

Figure 19. LibreOffice Wiki

Viele Open-Source-Projekte stellen ihre Dokumentation und das integrierte Hilfesystem ihres Programms auch in einem Wiki zur Verfügung. Zudem werden alte Versionen einer Seite (*Revisionen*) für künftige Referenzen archiviert.

Obwohl Wikis auch Projektdateien speichern können, gibt es bessere Werkzeuge für diesen Zweck, wie wir noch sehen werden.

Bug und Issue Tracker

Ein häufig genutztes Werkzeug in einem Open-Source-Projekt sind *Bug Tracker*, auch *Issue Tracker* genannt. Um ein Problem in der Software zu melden oder eine neue Funktion vorzuschlagen, könnten Sie eine E-Mail an das Projekt schicken — über einen Bug Tracker lassen sich jedoch alle erforderlichen Informationen und Schritte zur Reproduktion des Problems in strukturierter Form übermitteln, was es den Entwicklern deutlich einfacher macht. Ein solch strukturierter Bericht wird als *Bug Report* bezeichnet.

Außerdem gehen die Informationen nicht verloren, und das Projekt vergisst das Problem nicht; es kann es der richtigen Person zuordnen und sehen, wie viele Fehler bearbeitet oder behoben werden.

Das LibreOffice-Projekt bietet einen Bug Tracker, zu dem jeder beitragen kann (https://bugs.documentfoundation.org).

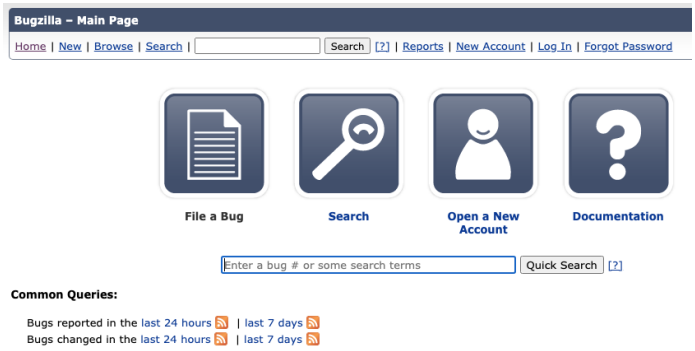

Figure 20. LibreOffice Bug Tracker

Helpdesks und Ticket-Systeme

Ein ähnliches Werkzeug ist ein *Helpdesk* oder *Ticket-System*, das weniger der Meldung von Softwareproblemen als der Unterstützung von Benutzern bei allen Arten von Problemen und Anfragen dient, beispielsweise zur Website des Projekts.

Der klassische Helpdesk ist eine Art Benutzer-Hotline: Der Benutzer meldet ein Problem, das in ein *Ticket* umgewandelt wird. Der Arbeitsablauf in einem Helpdesk-System dreht sich in der Regel um Prioritäten, Eskalationsstufen und Reaktionszeiten.

Nicht alle Open Source Communities bieten ein solches System an, aber viele kommerzielle Unternehmen haben eines. Für das LibreOffice-Projekt gibt es ein Ticket-System für die Infrastruktur, um Probleme mit Servern und Webdiensten zu melden.

Content Management System (CMS)

Ein weiteres wichtiges Tool für die Zusammenarbeit ist ein *Content Management System* (CMS). Wie der Name schon sagt, hilft es bei der Verwaltung von Inhalten, vor allem bei Websites. Wenn Sie zum Inhalt oder zur Gestaltung der Website eines Projekts beitragen wollen, sollten Sie sich mit dem CMS vertraut machen.

CMS helfen, ähnlich wie Wikis, Inhalte nach Kategorien und Sprachen zu strukturieren. Sie bieten oft einen WYSIWYG-Editor (“what you see is what you get”) und betten alles in ein passendes Template ein: Titel, Überschriften, Seitenlayout und Menüeinträge werden automatisch erstellt, so dass Sie sich ganz auf den Inhalt konzentrieren können. Außerdem verschwinden bei einer Umgestaltung der Seite die Inhalte nicht, sondern werden an das neue Template angepasst.

Document Management System (DMS)

Ein *Document Management System* (DMS) ist nicht zu verwechseln mit einem Content Management System: Während ein CMS dazu dient, Inhalte in einer definierten Vorlage darzustellen, etwa für die Präsentation einer Website, dient ein DMS der Verwaltung von Dokumenten wie Verträge, Rechnungen, Quittungen, E-Mails und allen anderen Arten von Korrespondenz.

Ein weiterer Unterschied besteht darin, dass ein CMS häufig für die öffentliche Präsentation verwendet wird, während ein DMS interne Dokumente verwaltet, die nicht für die Öffentlichkeit bestimmt sind.

Als Beiträger zu einem Open-Source-Projekt werden Sie wahrscheinlich seltener mit einem DMS in Berührung kommen, da es oft für bestimmte Rollen wie die Buchhaltung oder die Rechtsabteilung reserviert ist.

Source Code Management (SCM)

Als Entwickler stößt man in einem Open-Source-Projekt bald auf eines der wichtigsten Tools: die Plattform für das Source Code Management. Was das Wiki für Autren von Dokumentation ist das SCM-System für Softwareentwickler bei der gemeinsamen Arbeit am Code.

Zu den älteren SCM-Systemen gehören CVS und Subversion; heute kommt meist Git zum Einsatz, auch im LibreOffice-Projekt (https://git.libreoffice.org/). Die Werkzeuge sind auf der Kommandozeile verfügbar, aber es gibt auch grafische Schnittstellen, die die Interaktion insbesondere Anfängern erleichtern.

Source Code Management Systeme verfolgen die Versionen jeder Datei, verwalten Änderungen und Bearbeitungen des Codes, zeichnen auf, wer welche Änderung vorgenommen hat, und bieten im Idealfall einen vollständigen Überblick über die Entwicklung der Software.

Entwickler können einen bestimmten Zustand der Software “auschecken”, lokal daran arbeiten — einen Fehler beheben oder eine neue Funktion implementieren — und dann beantragen, dass diese Änderung in die Hauptentwicklungslinie der Software über einen sogenannten *Merge Request* (oder auch *Pull Request*) eingefügt wird. Die Annahme des Merge Requests führt die Änderungen des Autors mit dem Hauptcode zusammen.

Es gibt zentralisierte Plattformen (GitHub und GitLab), die SCM mit Wikis, Bug Tracking und anderen Tools für die Zusammenarbeit integrieren.

Source Code Management Systeme sind nicht auf Programmcode beschränkt: An dieser Lektion wurde zum Beispiel gemeinschaftlich in einem Git-Repository gearbeitet!

Dokumentation

Der Schlüssel zum Erfolg eines jeden Open-Source-Projekts ist eine saubere Dokumentation, idealerweise in mehreren Sprachen. Das LibreOffice-Projekt bietet aktuelle Bücher, Richtlinien und Referenzkarten (https://documentation.libreoffice.org) für seine Software sowie individuelle Hilfeseiten zu bestimmten Funktionen (https://help.libreoffice.org).

Es gibt verschiedene Arten von Dokumentation, auf die wir in den folgenden Abschnitten eingehen.

Dokumentation für Benutzer

Allgemein bekannt ist in der Regel die Dokumentation für Benutzer, die erklärt, wie die Software zu verwenden ist. Wenn Sie eine Eigenschaft oder Funktion des Programms nicht kennen, ist die Dokumentation — die von einzelnen Hilfeseiten bis zum Buch reichen kann — die erste Anlaufstelle.

Die Dokumentationswebsite, auf der die Handhabung der Software erklärt wird, ist neben der Produktwebsite, die einen Überblick über die Software und ihre Community bietet, oft eine der meistbesuchten Websites eines Projekts.

Dokumentation für Administratoren

Für den Einsatz in größeren Umgebungen, etwa in einem Unternehmen, enthält die Dokumentation für Administratoren alle relevanten Informationen, beispielsweise für die Verbindung mit Benutzerdatenbanken und Dateispeichern, die zentrale Konfigurationsverwaltung und die Handhabung von Updates.

Dokumentation für Entwickler und Architekten

Eine weitere Kategorie von Dokumentation richtet sich an Entwickler und Softwarearchitekten. Wenn Sie einen Beitrag zum Code eines Projekts leisten möchten, informiert Sie diese Dokumentation über die Softwarearchitektur, die Kodierungsstandards sowie Werkzeuge und Arbeitsabläufe, die bei der Arbeit an der Software zum Einsatz kommen.

Das LibreOffice-Projekt hat in seinem Wiki einen Leitfaden für Entwickler veröffentlicht, der Interessierten hilft, der Community beizutreten (https://wiki.documentfoundation.org/Documentation/DevGuide).

Geführte Übungen

1. Warum müssen sich gerade Open-Source-Projekte um geeignete Werkzeuge für Kommunikation und Zusammenarbeit kümmern?

2. Nennen Sie je ein Beispiel für synchrone und asynchrone Kommunikation.

3. Was ist ein Nachteil von Messenger-Apps und wie kann er vermieden werden?

4. Nennen Sie zwei Funktionen eines Wiki.

5. Was ist der Unterschied zwischen einem Bug Tracker und einem Helpdesk-System?

6. Was ist der Unterschied zwischen einem Content Management System und einem Document Management System?

7. Welchen Vorteil haben stand-alone oder föderale Systeme gegenüber zentralisierten Systemen?

Offene Übungen

1. Was ist einer der Hauptunterschiede zwischen einem lokalen Sportverein und einem internationalen Open-Source-Projekt?

2. Warum kann die Mitarbeit an einem Open-Source-Projekt besonders lohnend sein?

3. Welche Software setzt das Ubuntu-Projekt für das Bug Tracking ein?

4. Wie lautet der Name der Website für die Linux-Kernel-Mailinglisten?

Zusammenfassung

In dieser Lektion haben Sie eine Reihe von Werkzeugen kennengelernt, die für Kommunikation und Zusammenarbeit in einem Open-Source-Projekt eingesetzt werden. Sie haben den Unterschied zwischen synchroner und asynchroner Kommunikation sowie zwischen dezentralen, zentralen und stand-alone Lösungen kennengelernt. Sie haben auch erfahren, warum bestimmte Werkzeuge für bestimmte Aufgaben hilfreich sind und die Mitarbeit an einem Open-Source-Projekt erleichtern.

Antworten zu den geführten Übungen

1. Warum müssen sich gerade Open-Source-Projekte um geeignete Werkzeuge für Kommunikation und Zusammenarbeit kümmern?

   Die Zusammenarbeit in einer weltweit verteilten Gruppe bringt eine Reihe von Herausforderungen mit sich, die mit geeigneten Instrumenten zu bewältigen sind. Zudem bleiben die Freiwilligen vielleicht nicht ewig, so dass der Erhalt und die Weitergabe von Wissen ein weiterer wichtiger Aspekt beim Einsatz von Kommunikations- und Kollaborationswerkzeugen ist. Wenn Beiträge leicht einzubringen sind, trägt dies zur Nachhaltigkeit eines Projekts bei.

2. Nennen Sie je ein Beispiel für synchrone und asynchrone Kommunikation.

   Synchrone Kommunikation kann ein direktes Gespräch, ein Telefonanruf oder eine Videokonferenz sein. Beispiele für asynchrone Kommunikation sind E-Mail, SMS, Postbrief und Fax.

3. Was ist ein Nachteil von Messenger-Apps und wie kann er vermieden werden?

   Nach der Installation auf dem Smartphone erhalten Sie möglicherweise viele Benachrichtigungen — eine für jede neue Nachricht. Dies lässt sich durch die richtige Konfiguration vermeiden. Ein weiterer Nachteil ist, dass viele Messenger-Apps von proprietären Anbietern betrieben werden.

4. Nennen Sie zwei Funktionen eines Wiki.

   Gemeinschaftliche Bearbeitung und Übersetzung von Artikeln.

5. Was ist der Unterschied zwischen einem Bug Tracker und einem Helpdesk-System?

   Ein Bug Tracker ist ein spezielles Software Tool, um Fehler zu melden oder neue Funktionen in einer Software zu beantragen. Ein Helpdesk-System konzentriert sich auf die Unterstützung von Anfragen und die Verwaltung aller Arten von Problemen und Anfragen, beispielsweise auf der Website.

6. Was ist der Unterschied zwischen einem Content Management System und einem Document Management System?

   Ein CMS dient der meist öffentlichen Präsentation von Inhalten in einer vorgegebenen Form — meist einer Website. Ein DMS wird eingesetzt, um vorhandene Dokumente (meist intern) abzulegen.

7. Welchen Vorteil haben stand-alone oder föderale Systeme gegenüber zentralisierten Systemen?

   Ein zentralisiertes System steht unter der Kontrolle eines externen Anbieters. Alle Inhalte werden auf den Servern dieses Anbieters gespeichert und unterliegen dessen Geschäftsbedingungen.

Antworten zu den offenen Übungen

1. Was ist einer der Hauptunterschiede zwischen einem lokalen Sportverein und einem internationalen Open-Source-Projekt?

   Open-Source-Projekte sind nicht an eine bestimmte Sprache oder einen bestimmten Ort gebunden. Die Mitwirkenden können in verschiedenen Ländern und Kontinenten zuhause sein, unterschiedliche Sprachen sprechen und in verschiedenen Zeitzonen leben. Die meisten Aktivitäten im Rahmen eines Open-Source-Projekts finden virtuell statt, nicht persönlich.

2. Warum kann die Mitarbeit an einem Open-Source-Projekt besonders lohnend sein?

   Viele Open-Source-Projekte sind sehr groß und haben eine diverse Mitgliederstruktur. Durch die Zusammenarbeit können Sie vieles lernen, neue Dinge entdecken und Ihren Horizont erweitern.

3. Welche Software setzt das Ubuntu-Projekt für das Bug Tracking ein?

   Launchpad

4. Wie lautet der Name der Website für die Linux-Kernel-Mailinglisten?

   `https://lkml.org/`

Impressum

PDF generiert: 2025-02-18

Die LPI-Lernmaterialien sind eine Initiative des Linux Professional Institute (https://lpi.org). Die Lernmaterialien und ihre Übersetzungen finden Sie unter https://learning.lpi.org.

Für Fragen und Kommentare zu dieser Ausgabe sowie zum gesamten Projekt schreiben Sie eine E-Mail an: learning@lpi.org.