Teil I

Grundlagen

1 Application Programming Interfaces – eine Einführung

APIs sind keine neue Erfindung; ihre grundlegenden Prinzipien wurden bereits Mitte des 20. Jahrhunderts erkannt und seither in verschiedenen Formen genutzt. Ab der Jahrtausendwende nahmen Web-APIs an Bedeutung zu, und es entstand eine florierende API-Industrie, die Bereiche wie E-Commerce, soziale Medien, Cloud Computing und mobile Anwendungen prägt. In diesem Kapitel befassen wir uns zunächst mit der Entstehung der ersten APIs, definieren den Begriff und erläutern die Vorteile von APIs.

1.1 Eine kurze Geschichte der APIs

Das Konzept einer Subroutinen-Bibliothek wird erstmalig 1948 von Herman Goldstine und John von Neumann beschrieben [Goldstine & von Neumann 1948]. Demnach ist die Idee, dass die meisten Programme allgemeine Operationen wiederverwenden, um den Umfang von neuem Code und Fehlern zu reduzieren, nach Informatikermaßstäben schon sehr alt. Für die Idee der Subroutinen-Bibliotheken wurde Maurice Vincent Wilkes 1967 sogar mit dem Turing Award ausgezeichnet.

Wilkes und sein Team bauten den EDSAC-Röhrencomputer, der erstmals die Von-Neumann-Architektur implementierte und gespeicherte Programme ausführte. Wilkes damaliger Ph.D.-Student David Wheeler entwickelte für EDSAC ein detailliertes Schema zum Einsatz von Subroutinen. Während Goldstine und von Neumann vorsahen, das gesamte Programm in den Speicher zu laden und die Adressen vor Ausführung mit einer speziellen Routine zu ändern, entwickelte Wheeler eine Reihe von Initiierungsbefehlen, die zuerst ausgeführt wurden, um ein Programm von Lochkarten einzulesen und ohne weitere manuelle Eingriffe auszuführen. Die Initiierungsbefehle von Wheeler waren eine Art Boot Loader für die Programme auf den Lochkarten. Die Programme wurden in Assembler geschrieben, sodass die Benutzer des Computers nie mit dem Binärcode des Computers zu tun hatten.

Der technische Bericht »The preparation of programs for an electronic digital computer«, den das Team 1951 veröffentlichte, war ein Standardwerk der Programmierung, bis Jahre später die ersten höheren Programmiersprachen folgten. Wheeler veröffentlichte 1952 auf nur zwei Seiten folgende grundlegende Konzepte [Wheeler 1952]:

- Subroutinen
- Subroutinen-Bibliotheken
- Bedeutung von Dokumentation für Subroutinen-Bibliotheken
- Geheimnisprinzip
- Trade-off zwischen Generalität und Performance
- Funktionen höherer Ordnung
- Debugger
- Routinen zur Interpretierung von Pseudocode

In diesem Dokument schreibt Wheeler, dass die Vorbereitungen für eine Subroutinen-Bibliothek größer sind als ihre eigentliche Programmierung. Außerdem betont er die Bedeutung von Dokumentation von Subroutinen-Bibliotheken. Im abschließenden Fazit nennt er einfache Benutzung, Korrektheit und akkurate Dokumentation als Hauptziele bei der Konstruktion von Bibliotheken. Komplexität sollte vor Benutzern verborgen bleiben.

Obwohl Wheeler schon die Prinzipien der späteren APIs erkannte, unterschied er nicht zwischen API und Implementierung, denn es gab zu diesem Zeitpunkt nur eine Maschinenarchitektur und keine alternativen Implementierungen der Bibliotheken. Erst als die Bibliotheken wegen neuerer Hardware oder wegen besserer Algorithmen neu implementiert wurden und man existierende Programme portieren wollte, gab es Gründe, zwischen API und Implementierung zu unterscheiden.

1968 erschien erstmalig der Begriff »Application Programming Interface« [Cotton & Greatorex 1968]. API und Implementierung werden konzeptionell voneinander getrennt, um Implementierungen austauschen zu können, ohne dass Clients davon betroffen sind.

Mehr als ein halbes Jahrhundert nach Wheelers Pionierarbeit sind seine Aussagen immer noch gültig. Lediglich die Begriffe haben sich verändert. So schreibt Joshua Bloch [Bloch 2006], dass es einfach sein sollte, eine API korrekt zu benutzen, und dass es schwer sein sollte, eine API falsch zu benutzen. Egal wie gut eine API ist, ohne gute Dokumentation wird sie nicht benutzt.

1.2 Web-APIs ab dem Jahr 2000

Zur Jahrtausendwende begann die Suche nach innovativen Lösungen, um Produkte mehrerer E-Commerce-Webseiten miteinander zu verbinden. Web-APIs auf Basis der existierenden HTTP-Infrastruktur schienen das richtige Werkzeug für diese Aufgabe zu sein:

Erste XML-APIs von Salesforce.com

- Im Februar 2000 startete Salesforce.com offiziell eine webbasierte Sales Force Automation für Unternehmen. Dieser Internetdienst setzte von Anfang an XML-APIs ein. Salesforce.com reagierte damit auf den Kundenbedarf, Informationen zwischen verschiedenen Geschäftsanwendungen austauschen zu wollen.

Pionierarbeit von eBay

- Im November 2000, also nur sieben Monate nach Salesforce.com, ging die eBay-API zusammen mit dem eBay Developers Program live. Die eBay-API war eine Reaktion des Unternehmens auf die wachsende Anzahl an Applikationen, die bereits die eBay-Webseite benutzten. Die API sollte die Integration mit diesen und zukünftigen Applikationen vereinheitlichen. eBay kann deswegen als führender Pionier der Web-APIs und Webservices angesehen werden.

Soziale Medien

- Neben diesen E-Commerce-Plattformen spielten auch soziale Medien eine wichtige Rolle in der Geschichte der Web-APIs. 2003 startete del.icio.us, ein Bookmarking-Dienst zum Speichern, Teilen und Auffinden von Bookmarks für Webseiten. Mit einem leicht verständlichen URL-Schema[1] konnte man eine Liste mit Bookmarks für ein Schlüsselwort abrufen. Diese API war nahtlos in die Webseite integriert. Del.icio.us war eine der ersten Webseiten, die HTML zusammen mit maschinenlesbaren Inhalten wie RSS und XML anbot.

Web 2.0

- 2004 startete Flickr sein Webportal zum Hochladen, Kommentieren und Teilen von Bildern und kurzen Videos. Die Einführung einer RESTful API half Flickr, schnell populär für Blogger und Benutzer sozialer Medien zu werden. Flickr etablierte für Anwendungsentwickler zur Benutzung der API ein Self-Service. Neben seiner technischen Funktion wurde die API ein wichtiger Faktor für die weitere Geschäftsentwicklung. Die moderne Plattform von Flickr zählte zu den typischen Vertretern des »Web 2.0«.

1. Bookmark-Listen konnten für bestimmte Tags mit der HTTP-Methode GET abgerufen werden: XML-Inhalte mit *http://del.icio.us/api/tag/[tag_name]*, RSS mit *http://del.icio.us/rss/tag/[tag_name]* und HTML mit *http://del.icio.us/tag/[tag_name]*.

Facebook REST-API

- Die Entwicklungsplattform und API von Facebook ist seit 2006 verfügbar. Seitdem ist es Softwareentwicklern möglich, auf Facebook-Freunde, Fotos und Profilinformationen zuzugreifen. Die REST-API war ein Vorteil von Facebook gegenüber Konkurrenten wie MySpace.

Plattform X

- Im selben Jahr führte X seine API ein, basierend auf REST mit Unterstützung für JSON und XML. Ähnlich wie eBay reagierte X damit auf die wachsende Zahl von Applikationen. Die X-API wird von unzähligen Desktop-Clients, mobilen Anwendungen und sogar Geschäftsanwendungen genutzt.

Google Maps API

- Ebenfalls 2006 startete Google seine Google Maps API für die zahllosen Entwickler, die Google Maps in ihre Anwendungen integrieren wollten. Dies war die Geburtsstunde der Mashups, die neue Inhalte durch die Kombination bereits bestehender Inhalte erzeugen. Hierfür nutzen Mashups offene APIs, die von anderen Webanwendungen zur Verfügung gestellt werden.

API-Serviceprovider

- Die Liste bekannter Web-APIs ließe sich leicht fortsetzen. Wichtig sind ebenfalls die API-Serviceprovider wie Mashery. Dies war 2006 der erste Anbieter einer Infrastruktur zur Entwicklung, Veröffentlichung und Verwaltung von APIs, die es externen Entwicklern ermöglicht, Inhalte anderer Unternehmen für ihre Produkte zu nutzen.

Public Cloud PaaS

- In diesem Zeitraum begann außerdem die Ära des Public Cloud Computing Platform as a Service durch die Veröffentlichung der Amazon Web Services (AWS). Amazon startete mit dem Cloud-Speicher Amazon S3 und legte mit Amazon EC2, einem Webservice für die Bereitstellung von skalierbarer Rechenkapazität, nach. Beide bieten eine Web-API. PaaS sollte ein wichtiger Motor der API-Industrie werden.

Mobile Apps

- Foursquare startete 2009 einen standortgebundenen Dienst für mobile Geräte, mit dem Benutzer interessante Orte einer Stadt finden können. 2011 folgte die offizielle API von Instagram. APIs entwickelten sich vom Antreiber für E-Commerce-Anwendungen, soziale Medien und Cloud Computing zum Lieferanten von Ressourcen und Funktionen für mobile Geräte.

Versionierung

- Viele APIs haben eine lange Lebensdauer und werden daher versioniert. Die X-API v1, die 2006 online ging, wurde beispielsweise erst 2013 eingestellt. Zuvor gab es bereits eine neuere Version der API, sodass Entwickler ausreichend Zeit hatten, ihre Anwendungen anzupassen. Einige APIs haben praktisch eine unbegrenzte Lebensdauer. In der Java-Standardbibliothek wurden lange Zeit keine Elemente entfernt, sondern lediglich als »deprecated« gekennzeichnet. Besonders im Zuge der Einführung von Java 9 und des Modul-Systems hat

Oracle jedoch veraltete APIs entfernt. Auch in Java 11 wurden beispielsweise Java-Applets und das CORBA-Modell gestrichen, um die Java Development Kit (JDK) API schlanker und effizienter zu gestalten.

Internet der Dinge

- Das nächste große Kapitel der API-Geschichte trägt den Titel »Internet der Dinge« (Internet of Things – IoT). Hinter diesem Trend steckt die Idee, unterschiedlichste intelligente Geräte zu vernetzen, um Menschen bei ihren Tätigkeiten zu unterstützen. Das Spektrum dieser intelligenten Geräte reicht von Kühlschränken bis zu Autos. Mithilfe verschiedener APIs kann man auf die IoT-Einheiten zugreifen. Dies geschieht häufig drahtlos per Wi-Fi, BLE (Bluetooth Low Energy) oder NFC (Near Field Communication). Beispiele sind ThingSpeak, eine Open-Source-Lösung, mit der Entwickler unter Zuhilfenahme von Webtechnologien mit Geräten interagieren können, und OGC SensorThings API, ein Standard zum einheitlichen Zugriff auf IoT-Einheiten, Daten und Applikationen über das Web.

Microservices und Cloud-Architekturen

- Microservices und Cloud-Architekturen haben sich zwischen 2010 und 2020 als feste Bestandteile der IT-Landschaft etabliert. Das Konzept der Microservices entstand um 2010 und wurde durch Unternehmen wie Netflix und Amazon populär gemacht. Diese setzten auf kleine, unabhängige Dienste, um ihre Anwendungen effizient bereitzustellen. In den darauffolgenden Jahren gewannen Microservices zunehmend an Bedeutung. Viele Unternehmen übernahmen diese Architektur, da sie mehr Agilität, Skalierbarkeit und eine einfachere Wartung ihrer Software ermöglichten. Zeitgleich wurden Cloud-Architekturen von zahlreichen Organisationen adaptiert, um Kosten zu senken und die Bereitstellung von Anwendungen zu beschleunigen. Insgesamt haben Microservices und Cloud-Architekturen den Softwareentwicklungsprozess grundlegend verändert. APIs spielen dabei eine Schlüsselrolle, da sie die Interoperabilität und Modularität in diesen modernen Architekturen maßgeblich unterstützen.

1.3 API-Definition

Eine API ist eine Schnittstelle, die es zwei Softwarekomponenten innerhalb einer oder mehrerer Anwendungen oder Systeme ermöglicht, miteinander zu kommunizieren und Daten auszutauschen, ohne dass die interne Funktionsweise der jeweiligen Komponenten offengelegt wird. Eine API kann als eine Art »Vertrag« betrachtet werden, der festlegt, welche Funktionen oder Daten eine Softwarekomponente bereitstellt und wie andere Softwarekomponenten diese anfordern oder nutzen können. Dieser »Vertrag« umfasst sowohl die technische Definition der Schnittstellenfunktionen als auch eine Abstraktion der zugrunde liegenden Problemlösung. Er legt fest, wie Nutzer mit den Komponenten interagieren sollten, die eine Lösung für das betreffende Problem implementieren [Reddy 2011].

Zwei wesentliche Aspekte einer API lassen sich unterscheiden:

- Technische Definition: Diese beschreibt die genauen Anforderungen und Spezifikationen für Eingaben, Ausgaben und Methoden, damit eine Anwendung erfolgreich mit einer anderen interagieren kann. Dazu gehören oft festgelegte Datenformate, Authentifizierungsmechanismen und Endpunkte. Eine API erfordert in der Regel eine detaillierte Dokumentation dieser technischen Spezifikationen, um ihre Nutzung zu ermöglichen.
- Abstraktionsebene: APIs kapseln im Sinne des Geheimnisprinzips die Komplexität des zugrunde liegenden Systems und stellen nach außen hin nur die benötigten Funktionen bereit. Das bedeutet, dass Entwickler nicht verstehen müssen, wie der Code hinter der API implementiert ist, um sie nutzen zu können. Diese Abstraktion erleichtert die Integration und Nutzung unterschiedlicher Systemkomponenten.

Allgemeine API-Definition

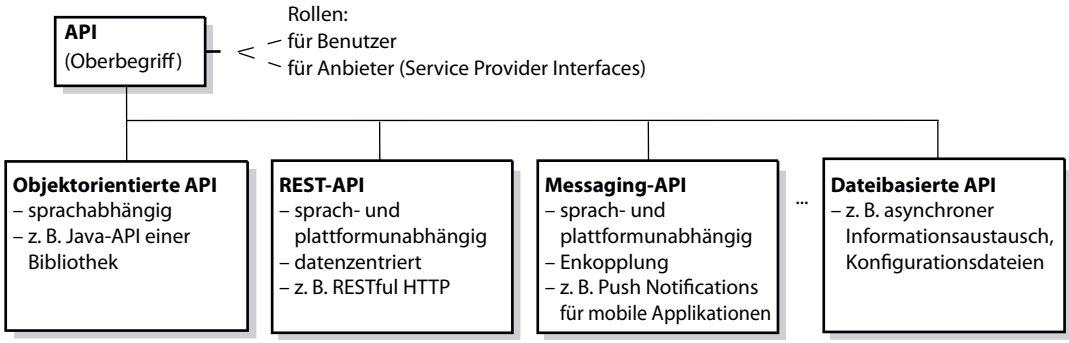
Joshua Bloch definiert APIs folgendermaßen: »Eine API spezifiziert die Operationen sowie die Ein- und Ausgaben einer Softwarekomponente. Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass die Implementierung variieren kann, ohne die Nutzer der Komponente zu beeinträchtigen« [Bloch 2014].

API-Typen in diesem Buch

Diese allgemeine Definition verdeutlicht, dass »API« nur der Oberbegriff für viele unterschiedliche API-Spielarten ist. Konkret unterscheidet dieses Buch zwischen Programmiersprachen-APIs und Remote-APIs:

- Programmiersprachen-APIs werden beispielsweise von Bibliotheken angeboten und sind sprach- und plattformabhängig. Als Vertreter der Programmiersprachen-APIs behandelt das Buch objektorientierte Java-APIs.
- Auf der Seite der Remote-APIs bietet das Buch RESTful HTTP, SOAP-Webservices und Messaging-APIs. Diese APIs sind durch Protokolle wie HTTP sprach- und plattformunabhängig. Diese Eigenschaften erfüllen ebenfalls SOAP-Webservices, die mit und ohne HTTP (z. B. über eine Message Queue) genutzt werden können. Messaging-APIs bieten asynchrone Kommunikation auf Protokollen wie AMQP (Advanced Message Queuing Protocol) oder MQTT (Message Queue Telemetry Transport). Darüber hinaus gibt es auch Remote Procedure Calls (RPCs) und dateibasierte APIs für Konfigurationen und asynchronen Informationsaustausch.

Abb. 1–1
Es gibt verschiedene API-Typen. In diesem Buch werden objektorientierte Java-APIs, Web-APIs und Messaging-APIs behandelt.



Service Provider Interface

Auch Frameworks haben eine API, über die sie benutzt und erweitert werden können. Ein Service Provider Interface (SPI) ist eine API, die dazu bestimmt ist, von einem Benutzer erweitert oder implementiert zu werden. Auf diese Weise kann eine Applikation oder ein Framework Erweiterungspunkte bereitstellen. Generell ist für Programmiersprachen-APIs das Thema Vererbung wichtig.

API versus Protokoll

Häufig werden APIs und Protokolle im gleichen Kontext verwendet. Dennoch können und sollten die Begriffe voneinander getrennt werden.

Eine objektorientierte API kann ein Protokoll kapseln. Ein Beispiel dazu ist im Java-Umfeld RMI (Remote Method Invocation). Eine Implementierung dieser Java-API nutzt intern das Protokoll JRMP (Java Remote Method Protocol) für entfernte Aufrufe zwischen Objekten. Ein anderes Beispiel ist JMS (Java Messaging Service). Diese Java-API für Message Queues wird von einem JMS-Provider (z. B. ActiveMQ) mithilfe eines zugrunde liegenden Protokolls (z. B. AMQP) umgesetzt. Eine API-Implementierung kann demzufolge ein Protokoll kapseln bzw. es

implementieren. Umgekehrt gilt das nicht: Ein Protokoll kann keine API kapseln oder implementieren.

HTTP ist ein Protokoll, das ist unstrittig, aber HTTP allein stellt noch keine API dar. Eine API kann man jedoch als eine Menge von HTTP-Requests und -Responses inklusive der Struktur der verwendeten Nachrichten definieren. Häufig spricht man in diesem Fall von einer Web-API oder allgemein von einer Remote-API.

Ein anderes Beispiel sind WebSockets, weil hier Protokoll und API konsequent voneinander getrennt sind. Das WebSocket-Protokoll ist in RFC 6455 [Fette & Melnikov 2011] standardisiert und spezifiziert u. a. das Öffnen und Schließen von Verbindungen mit Handshake. Die WebSocket-API [Hickson 2011] definiert u. a. ein WebSocket-Interface mit den Methoden send und close. Mithilfe der API können Webseiten das WebSocket-Protokoll für die Zwei-Wege-Kommunikation mit einem entfernten Host nutzen. Moderne Webbrowser exponieren die API und nutzen das Protokoll zur Kommunikation mit entfernten Servern, die das Protokoll möglicherweise hinter einer serverseitigen API anbinden.

1.4 Vorteile einer API

In der zuvor betrachteten kurzen Geschichte der APIs wurden bereits einige Vorteile beschrieben. Ein wichtiger Vorteil, der sich durch die Trennung zwischen API und Implementierung ergibt, ist die Änderbarkeit oder Austauschbarkeit der Implementierung. Solange der Vertrag der API eingehalten wird, müssen Benutzer ihren Code nicht anpassen. Aus diesem Ansatz ergeben sich mehrere Vorteile:

Stabilität durch lose Kopplung

- Angenommen, Benutzer einer Softwarekomponente wären direkt von den Implementierungsdetails der Softwarekomponente abhängig, dann wäre der Code des Benutzers instabil, weil dieser schon bei kleinen Änderungen der Softwarekomponente angepasst werden müsste. Diese starke Kopplung zwischen einer Softwarekomponente und ihren Benutzern kann durch eine API minimiert werden. Falls beispielsweise ein Webservice seine Ein-Server-Lösung durch eine Lösung mit verteilter Architektur ersetzen muss, weil die Benutzeranzahl wächst und Performanceprobleme auftreten, sollte die Änderung keine Auswirkungen auf bestehende Benutzer haben, obwohl die neue Lösung auf einer völlig anderen Architektur basiert.

- Beispielsweise kann ein in ANSI C geschriebenes Programm auf verschiedenen Computerarchitekturen und Betriebssystemen ausgeführt werden, sofern eine konforme C-Implementierung vorhanden ist. Als Beispiel könnte man auch die Java Runtime Environment (JRE) nennen, denn sie bietet für Java-Programme eine einheitliche API-Implementierung für verschiedene Betriebssysteme.

Portabilität

- Eine API bietet eine geeignete Abstraktion und versteckt die Komplexität der Implementierung. Diese nicht notwendige Kenntnis des API-Benutzers über Implementierungsdetails folgt dem Geheimnisprinzip und hilft, die Komplexität großer Anwendungen zu beherrschen. Die Modularisierung hat wiederum Vorteile für Arbeitsteilung und Entwicklungskosten.

Komplexitätsreduktion durch Modularisierung

- Eine API wird nicht nur entworfen, um Implementierungsdetails zu verbergen, sondern um Funktionen einer Softwarekomponente anderen Entwicklern möglichst einfach zur Verfügung zu stellen. Aus diesem Grund sollte eine API für einfache Wiederverwendung und Integration optimiert werden. Mit RESTful HTTP kann beispielsweise eine einheitliche Schnittstelle für unterschiedliche Web-APIs realisiert werden.

Softwarewiederverwendung und Integration

Grundvoraussetzung für die genannten Vorteile ist gutes API-Design. Allerdings ist es gar nicht so einfach, gute APIs zu entwerfen. Das Entwerfen schlechter APIs geht vergleichsweise leicht [Henning 2007]. Eine gute API erkennt man sofort, sobald man sie verwendet: Ihre Benutzung macht Spaß, und es gibt kaum Reibungsverluste, weil sie intuitiv benutzbar und gut dokumentiert ist.

Die Konsequenzen schlechten API-Designs sind vielfältig und schwerwiegend: Schlechte APIs sind schwer zu benutzen, und in manchen Fällen muss zusätzlicher Clientcode geschrieben werden, der Programme größer, komplizierter und schwerer wartbar macht. Entwickler brauchen mehr Zeit, um schlechte APIs zu verstehen und zu benutzen. Schlechte APIs führen deswegen zu erhöhten Entwicklungskosten oder zur völligen Ablehnung von Softwarekomponenten, falls Entwickler zwischen mehreren Alternativen wählen können.

1.5 Nachteile einer API

Interoperabilität

Eine API hat sicherlich nicht nur Vorteile, sondern auch Nachteile. Erwähnenswert ist die fehlende Interoperabilität von Programmiersprachen-APIs, denn eine Java-API kann beispielsweise nicht von einer Go-Applikation konsumiert werden. Die Lösung liegt allerdings auf der Hand: Die gewünschte Interoperabilität bieten Remote-APIs auf Basis von Protokollen wie HTTP und AMQP, weil diese von unterschiedlichen Plattformen und Programmiersprachen genutzt werden können.

Änderbarkeit

Ein anderer Nachteil, der im Alltag Kopfschmerzen bereiten kann, ist die eingeschränkte Änderbarkeit von APIs, denn die mit den API-Benutzern geschlossenen API-Verträge dürfen nicht gebrochen werden – oder etwa doch?

Um das Problem besser erläutern zu können, sollte man zwischen interner und veröffentlichter API unterscheiden. Letztere hat Benutzer, die Sie nicht kennen oder die Sie nicht kontrollieren. In diesem Fall dürfen Sie keine Änderungen machen, die den bestehenden API-Vertrag brechen. Für die internen APIs ist die Situation anders: Wenn Sie zum Beispiel Ihre Codebasis in Module mit öffentlichen und privaten Teilen strukturieren, erfolgt die Kommunikation der Module untereinander über deren öffentliche Teile, also über deren APIs. In diesem Fall können Sie die APIs ändern und von Refactoring profitieren, weil Sie den von den APIs abhängigen Code kontrollieren. Generell sollten Sie so wenig wie möglich veröffentlichen, um Änderungen machen zu können.

1.6 API als Produkt

Die bisher genannten Funktionen sind hauptsächlich technischer Art. Es gibt jedoch auch wirtschaftliche Funktionen, die wir hier nicht vergessen dürfen. Prinzipiell kann man in diesem Zusammenhang zwischen zwei Unternehmenstypen unterscheiden:

Das Unternehmen ist die API.

- Für Unternehmen wie Yelp, Twilio oder SendGrid ist die API das Hauptprodukt. Das Bereitstellen einer nützlichen und leicht verwendbaren API ist ihre Geschäftsgrundlage.

Die API als zusätzlicher Kanal

- Für andere Unternehmen wie FedEx, Walmart und Best Buy steht die API weniger im Mittelpunkt. Viele Offline-Unternehmen nutzen APIs, um ihren Markt zu vergrößern.

1.7 Welche Strategien verfolgen Unternehmen mit APIs?

Web-APIs können für Unternehmen von strategischer Bedeutung sein. Einige dieser Strategien sind hier aufgezählt:

- Unternehmen wie Facebook können nicht für jedes mobile Gerät einen dedizierten Client bauen. Stattdessen setzen sie eine neutrale API ein, die ausfallsicher und skalierbar ist. *Mobile Strategie*
- X konnte die Benutzung seiner Plattform durch eine besonders gute API steigern. Denn diese API war die Grundvoraussetzung für die vielen mobilen Apps, mit denen X auf unzähligen Geräten überall und jederzeit genutzt werden kann. *Benutzung der Plattform antreiben*
- Eine API-Strategie kann der Anfang eines neuen Geschäftszweigs sein und Wachstum zur Folge haben. Auch Best Buy startete eine API zur Steigerung seines Onlinehandels. Schnell entstanden Apps zum Preisvergleich, zum Stöbern im Warenkatalog und zur Verbreitung von Angeboten. *Investition in neue Geschäftszweige*
- APIs sind wichtig zur Vernetzung mit Partnern und Zulieferern. Beispielsweise verfolgt FedEx eine API-Strategie, durch die unzählige FedEx-kompatible Applikationen entstanden sind. *Integration mit Partnern*
- APIs dienen nicht nur zur Integration von Fremdsystemen, sondern auch innerhalb eines Unternehmens sind sie zur Integration von Systemen wichtig. *Integration innerhalb eines Unternehmens*

1.8 API-first-Ansatz

Vielleicht hatten Sie in der Praxis schon einmal mit einer Legacy-Anwendung zu tun, bei der APIs nachträglich hinzugefügt wurden. In diesem Fall könnte man von einem »API-last«-Ansatz sprechen. Nachträglich hinzugefügte APIs basieren oft auf vorhandenen Backend-Logiken und Datenmodellen, was ihre Flexibilität und Wiederverwendbarkeit einschränkt, da sie eng mit der bestehenden Architektur verknüpft sind. Eine nachträgliche API-Integration kann eingeschränkt sein, weil das zugrunde liegende System nicht darauf ausgelegt ist, über APIs zu kommunizieren. Das Backend kann möglicherweise nicht alle erforderlichen Daten effizient bereitstellen, was die Funktionalität und das Design der API einschränken könnte. *API-last*

Um diese Nachteile zu vermeiden, hat sich der API-first-Ansatz etabliert. Die Grundidee ist wie folgt: Im Softwareentwicklungsprozess werden APIs als zentrale Bausteine der Software betrachtet und entsprechend priorisiert. APIs sind nicht nur Middleware, Klebstoff oder schlimmer noch ein nachträglicher Einfall, sondern bewusste Strategie. *API-first*

Nimmt man den Begriff »API-first« wörtlich, so entwickeln Teams, die diesem Ansatz folgen, erst ihre APIs, bevor sie sich anderen Aspekten wie der Persistenz oder der Geschäftslogik im Backend widmen. Auf diese Weise entstehen Anwendungen, die auf internen und externen Diensten basieren, die über APIs bereitgestellt werden [Postman API-first].

Die Reihenfolge ist jedoch nicht zwingend entscheidend, sondern vielmehr eine Folge der Priorisierung von Entwicklung und Einsatz der APIs. API-first-Organisationen verstehen genau, dass Anwendungen als Verknüpfung interner und externer Dienste über APIs aufgebaut werden. Sie erkennen, dass APIs festlegen, wie digitale Ressourcen und Funktionen des Unternehmens internen und externen Nutzern zugänglich gemacht werden, um Geschäftsanwendungen bedarfsgerecht zu unterstützen.

API-first-Entwicklungsmodell einführen

Damit eine Organisation ein API-first-Entwicklungsmodell einführen kann, muss sie APIs priorisieren, die Rolle öffentlicher, privater und Partner-APIs in Organisationen erkennen und den API-Lebenszyklus und die Tools verstehen, die erforderlich sind, um API-first zu werden [Postman API-first].

Priorisierung von APIs

- API-first-Organisationen stellen die APIs, die ihre Anwendungen unterstützen, in den Mittelpunkt und fokussieren sich auf den Mehrwert, den diese APIs für das Unternehmen schaffen können, anstatt lediglich eine Anwendung zu entwickeln und die API nachträglich hinzuzufügen. Dieser vorausschauende Ansatz ermöglicht es, die Anwendung über die API in verschiedenen Bereichen des Unternehmens für vielfältige Zwecke einzusetzen. Außerdem erkennen API-first-Organisationen, dass APIs gewartet und verbessert werden müssen, und bilden Teams, die dies unterstützen.
- Tatsächlich sind die meisten APIs, die Entwickler verwenden, intern. Laut einer repräsentativen Umfrage von Postman sind 58 % der von den Befragten verwendeten APIs ausschließlich für interne Zwecke, 27 % für Partner und 15 % der APIs öffentlich im Web erreichbar [Postman API-first].
- Ein klar definierter API-Lebenszyklus ist entscheidend, um den Betrieb auf einer API-Plattform optimal zu gestalten und Hunderte oder sogar Tausende von APIs über verschiedene Teams hinweg effizient zu verwalten. Ein gemeinsames Verständnis des API-Lebenszyklus und ein einheitliches Vokabular für dessen Beschreibung unterstützen Ihre Teams dabei, abgestimmt zu arbeiten und APIs mit höherer Produktivität, Qualität und Governance zu entwickeln – genau die Voraussetzungen, die Ihr Unternehmen zukunftsfähig machen.

- API-Plattformen sind Softwaresysteme, die integrierte Tools und Prozesse bieten, um Entwicklern und Nutzern das Erstellen, Verwalten, Veröffentlichen und Verwenden von APIs zu ermöglichen. Mehr Informationen finden Sie in Kapitel 16.

1.9 Zusammenfassung

In diesem Kapitel haben Sie einen Überblick über die Geschichte der APIs bekommen. Hier sind die wichtigsten Etappen kurz zusammengefasst:

- Die ersten Subroutinen-Bibliotheken gab es 1949/1950 für den Supercomputer EDSAC.
- Der Begriff »API« wurde 1968 erstmals erwähnt.
- Ab 2000 entstehen erste Web-APIs für E-Commerce. Daraufhin entwickelt sich eine ganze API-Industrie für soziale Medien, Cloud Computing, mobile Applikationen und schließlich das Internet der Dinge.

In diesem Kapitel wurden ebenfalls wichtige Vorteile von APIs vorgestellt:

- Man kann konzeptionell zwischen einer API und ihrer Implementierung unterscheiden. Eine API bildet den Vertrag und die Beschreibung einer Softwarekomponente.
- Eine gute API ist für einfache Wiederverwendung und Integration optimiert.
- API-first-Organisationen rücken die APIs, die ihre Anwendungen unterstützen, in den Mittelpunkt. Der Fokus liegt dabei auf dem Mehrwert, den diese APIs für das Unternehmen schaffen, anstatt lediglich eine Anwendung zu entwickeln und die API nachträglich hinzuzufügen.

Das nächste Kapitel geht der Frage nach, was eine gute API ausmacht. Dazu werden Sie verschiedene Qualitätsmerkmale kennenlernen.

2 Qualitätsmerkmale

Nachdem Sie im vorherigen Kapitel die Prinzipien und den Zweck von APIs kennengelernt haben, geht es in diesem Kapitel weiter mit den allgemeinen Qualitätsmerkmalen. Diese Merkmale sind das Ziel der Best Practices und Design-Heuristiken in diesem Buch.

2.1 Allgemeine Qualitätsmerkmale

Um die Qualität eines Produktes oder einer Applikation bewerten zu können, gibt es viele Qualitätsmodelle, von denen sich insbesondere DIN/ISO 9126 in der Praxis durchgesetzt hat. Die darin definierten Qualitätsziele gelten für Software im Allgemeinen und damit auch für APIs. Ein Ziel ist beispielsweise die Richtigkeit der geforderten Funktionalität. Zweifellos ist das ein wichtiges Ziel, doch welche Ziele kann man für APIs besonders hervorheben?

- APIs sollen für andere Entwickler leicht verständlich, erlernbar und benutzbar sein. Gute Benutzbarkeit ist ein zentrales Ziel beim API-Design, deshalb finden Sie in diesem Kapitel weitere Informationen darüber. *Benutzbarkeit*
- Insbesondere für mobile Applikationen ist geringer Akku-Verbrauch und geringes Online-Datenvolumen wichtig. Remote-APIs und eventuell dazugehörige Software Development Kits (SDKs), mit denen die Remote-APIs aufgerufen werden, sollten dies berücksichtigen. Auch Skalierbarkeit kann ein wichtiges Ziel sein, falls Sie beispielsweise davon ausgehen, dass Ihre API in Zukunft immer häufiger aufgerufen wird. Kapitel 14 bietet weitere Informationen zu diesem Thema. *Effizienz*
- Die Reife einer API-Implementierung hängt von der Versagenshäufigkeit durch Fehlerzustände ab. Interessant für API-Designer ist vor allem die Frage, wie man mit Fehlern umgehen soll. Informationen über Exception Handling und zur Fehlerbehandlung von Web-APIs finden Sie in Abschnitt 5.8 und 9.4. *Zuverlässigkeit*

2.2 Benutzbarkeit

Wann ist eine API gut benutzbar? Vermutlich kann diese Frage nur mit einer subjektiven Einschätzung beantwortet werden. Dennoch gibt es eine Reihe allgemein akzeptierter Eigenschaften. Weil aber diese Eigenschaften in der Praxis nie vollständig umgesetzt werden können, könnte man auch von Zielen sprechen:

- konsistent
- intuitiv verständlich
- dokumentiert
- einprägsam und leicht zu lernen
- lesbaren Code fördernd
- schwer falsch zu benutzen
- minimal
- stabil
- einfach erweiterbar

Diese Eigenschaften werden in den folgenden Abschnitten vorgestellt.

2.2.1 Konsistent

Kohärentes Design mit der Handschrift eines Architekten

»Konsistenz« deckt sich weitestgehend mit »konzeptioneller Integrität«. Dieses Grundprinzip besagt, dass komplexe Systeme ein kohärentes Design mit der Handschrift eines Architekten haben sollten. Dieses Designprinzip stammt von Frederick Brooks, der bereits vor mehreren Jahrzehnten schrieb: »Konzeptionelle Geschlossenheit ist der Dreh- und Angelpunkt für die Qualität eines Produkts […]« [Brooks 2008]. Er meint damit, dass Entwurfsentscheidungen, wie beispielsweise Namensgebungen und die Verwendung von Mustern für ähnliche Aufgaben, im gesamten System durchgängig angewandt werden sollen. Das folgende Beispiel soll diese Aussage verdeutlichen. Zu sehen sind zwei Listen mit Funktionsnamen [Lacker 2013]:

```
str_repeat      strcmp
str_split       strlen
str_word_count  strrev
```

Die Liste auf der linken Seite beginnt mit einem Präfix »str«. Darauf folgen die Funktionsbezeichnungen, wobei die einzelnen Wörter durch Unterstriche voneinander getrennt sind. Die Funktionsnamen auf der rechten Seite sind ähnlich aufgebaut. Der Unterschied ist, dass man hier auf die Unterstriche verzichtet hat. Vermutlich haben Sie schon erkannt, dass es sich hierbei um die Bezeichnungen von Funktionen zur Bearbeitung von Zeichenketten handelt. Beide Namenskonventionen sind in Ordnung. Es ist eine Frage des persönlichen Geschmacks, welche man bevorzugt.

Was ist nun das Problem? Das Problem liegt darin, dass beide Namenskonventionen zur gleichen PHP-API gehören. Das bedeutet, dass sich Entwickler nicht nur die Namen der Funktionen, sondern auch ihre Namenskonvention merken müssen. Aus diesem Grund sollte eine API unbedingt die (nur eine) Handschrift eines Architekten[1] tragen.

Auch im Java Development Kit (JDK) lassen sich leicht Beispiele finden. Das Wort »Zip« wird im selben Package mal mit CamelCase und mal komplett in Großbuchstaben geschrieben:

```
java.util.zip.GZIPInputStream
java.util.zip.ZipOutputStream
```

Das Setzen des Textes eines Widgets ist nicht einheitlich im JDK gelöst. Mehrheitlich heißt die Methode setText, aber leider gibt es Abweichungen:

```
java.awt.TextField.setText();
java.awt.Label.setText();
javax.swing.AbstractButton.setText();
java.awt.Button.setLabel();
java.awt.Frame.setTitle();
```

2.2.2 Intuitiv verständlich

Die zweite wichtige Eigenschaft einer guten API ist intuitive Verständlichkeit. Eine intuitiv verständliche API ist in der Regel auch konsistent und verwendet einheitliche Namenskonventionen. Das bedeutet, dass gleiche Dinge die gleichen Namen haben. Und umgekehrt haben unterschiedliche Dinge auch unterschiedliche Namen. Dadurch ergibt sich eine gewisse Vorhersagbarkeit. Betrachten wir dazu ein weiteres Beispiel:

Ruby-Methoden mit Ausrufezeichen (!)

Ruby-Methoden, die mit einem Ausrufezeichen (!) enden, ändern das Objekt, auf dem sie aufgerufen wurden. Methoden ohne Ausrufezeichen am Namensende erzeugen hingegen eine neue Instanz und lassen das Objekt, auf dem sie aufgerufen wurden, unverändert.

```
my_string.capitalize
# Funktioniert wie capitalize, erzeugt aber keinen neuen String
my_string.capitalize!

my_string.reverse
# Funktioniert wie reverse, erzeugt aber keinen neuen String
my_string.reverse!
```

Nachdem Sie die Beispiele für capitalize! und reverse! gesehen haben, können Sie vermutlich das Namenspaar für »downcase« erraten.

1. Selbstverständlich könnte es auch die Handschrift einer Architektin sein.

Setter- und With-Methoden

Für Java gibt es ebenfalls derartige Konventionen. Eine Konvention betrifft Setter-Methoden wie setName, setId oder setProperty. Setter-Methoden ändern das aufgerufene Objekt. Methoden wie withName, withId oder withProperty ändern das aufgerufene Objekt nicht, sondern erzeugen ein neues Objekt mit den angegebenen Werten. Das Präfix »with« wird beispielsweise von der Bibliothek Joda-Time genutzt.

Methoden der Java-Collections

Ein anschauliches Beispiel für die Bedeutung konsistenter Terminologie in APIs sind die Collections der Java-Standardbibliothek. Hier wurden starke Begriffe wie »add«, »contains« und »remove« etabliert, die einheitlich in den Interfaces List und Set vorkommen. Im Gegensatz dazu weicht das Interface von Map jedoch ab. Warum ist das so? Wäre ein einheitliches »add« in allen drei Interfaces nicht sinnvoller gewesen?

Die folgende Tabelle zeigt die Interfaces von List, Set und Map im Vergleich:

java.util.List	java.util.Set	java.util.Map
add	add	put
addAll	addAll	putAll
contains	contains	containsKey, containsValue
containsAll	containsAll	–
remove	remove	remove
removeAll	removeAll	–

Die unterschiedliche Benennung der Methoden zum Hinzufügen von Elementen in den Schnittstellen List, Set und Map – add bei List und Set, put bei Map – spiegelt die unterschiedlichen Konzepte und Verwendungszwecke dieser Datenstrukturen wider.

- Mit add wird einem List- oder Set-Objekt einfach ein Element hinzugefügt. Diese Operation fügt ein neues Element in die Collection ein, ohne dabei eine explizite Zuordnung vorzunehmen.
- Die Methode put in einer Map ist komplexer, da sie zwei Objekte – einen Schlüssel und einen Wert – miteinander verknüpft und diese als Paar speichert. Der Ausdruck »put« (dt. »setzen«, »stellen« oder »legen«) signalisiert das Einfügen oder Ersetzen eines Werts für einen bestimmten Schlüssel.

Diese Unterscheidung in den Methodennamen sorgt für mehr Klarheit und vermeidet Missverständnisse in Bezug auf die Funktionsweise der jeweiligen Datenstruktur.

Fehlen von removeAll in Map

Es mag auf den ersten Blick seltsam erscheinen, dass Map keine Methode removeAll hat, die das Gegenstück zu putAll wäre. Allerdings

ist das Verhalten von removeAll in einer Map potenziell mehrdeutig. Soll es alle Einträge entfernen, deren Schlüssel in der übergebenen Collection enthalten sind? Oder sollen die Einträge entfernt werden, deren Werte in der Collection vorkommen?

Diese Mehrdeutigkeit könnte zu Verwirrung führen, und daher verzichtet Java auf eine generelle removeAll-Methode für Map. Stattdessen bietet die API spezifische Methoden:

```
// Entfernen von Einträgen anhand übereinstimmender Schlüssel
map.keySet().removeAll(someCollection);
// Entfernen von Einträgen anhand übereinstimmender Werte
map.values().removeAll(someCollection);
// Selektives Entfernen mit beliebiger Bedingung
map.entrySet().removeIf(entry ->
conditionBasedOnKeyOrValue(entry));
```

Konsistenz in der Benennung

In Map finden wir ein weiteres Beispiel für die Bedeutung konsistenter Begriffe: Während Map über entrySet und keySet verfügt, gibt es kein valueSet. Stattdessen heißt diese Methode korrekt values, da eine Menge (Set) der Definition nach keine doppelten Elemente enthält, während Werte in einer Map durchaus doppelt vorkommen können.

Synonyme vermeiden

Es ist entscheidend, starke Begriffe in einer API zu etablieren und diese konsequent wiederzuverwenden. Innerhalb einer API sollten Synonyme wie »delete« und »remove« nicht vermischt werden. Entscheiden Sie sich für einen Begriff und halten Sie ihn durchgängig ein. Verwenden Sie Begriffe, die für die Benutzer der API vertraut sind. Beispielsweise wäre »erase« zu ungewohnt; ein Java-Entwickler würde in Dateioperationen eher nach »create« und »delete« suchen, während bei Datenbankoperationen »insert« und »remove« gängiger wären. Für Builder sind Methoden wie »withX« oder »addY« üblich, um Objekte schrittweise zu erstellen. Je nach Kontext können auch Methoden mit dem Präfix »plus« verwendet werden, wie etwa date.plusDays(3), um eine neue Instanz mit aktualisierten Werten zu erzeugen.

Intuitive APIs

Eine API ist intuitiv, wenn Entwickler ihren Clientcode auch ohne Dokumentation verstehen können. Dies gelingt nur durch sprechende Bezeichner und das Wiederverwenden bekannter Begriffe aus anderen gängigen APIs. Es ist daher ratsam, etablierte Terminologie aus bekannten Bibliotheken und Frameworks zu übernehmen, um das Verständnis zu erleichtern.

2.2.3 Dokumentiert

Eine API sollte möglichst einfach zu benutzen sein. Gute Dokumentation ist für dieses Ziel unverzichtbar. Neben Erklärungen für einzelne

Klassen, Methoden und Parameter sollten auch Beispiele in der Dokumentation vorhanden sein. Entwickler können durch Beispiele schnell eine API lernen und benutzen. Im Idealfall findet ein Entwickler ein passendes Beispiel, das mit wenigen Änderungen direkt wiederverwendet werden kann. Die Beispiele der Dokumentation zeigen, wie die API korrekt verwendet werden soll.

Gute Dokumentation kann zum Erfolg einer Technologie beitragen. Das Spring Framework hat beispielsweise eine sehr gute Dokumentation mit vielen sinnvollen Beispielen und Erklärungen. Dies war sicherlich ein Grund für die hohe Akzeptanz des Frameworks.

2.2.4 Einprägsam und leicht zu lernen

Wie leicht oder schwer es ist, eine API zu lernen, hängt von vielen unterschiedlichen Faktoren ab. Eine konsistente, intuitiv verständliche und dokumentierte API ist sicherlich einfacher zu lernen als eine inkonsistente, unverständliche und undokumentierte. Die Anzahl der von einer API verwendeten Konzepte, die Wahl der Bezeichner und das individuelle Vorwissen der Benutzer haben ebenfalls großen Einfluss auf die Lernkurve.

APIs sind nur mit Mühe zu erlernen, wenn die Einstiegshürden sehr hoch gelegt werden. Dies ist dann der Fall, wenn viel Code für erste kleine Ergebnisse geschrieben werden muss. Nichts kann einen Benutzer mit Anfängerkenntnissen mehr einschüchtern. Das Webframework Vaadin bietet deswegen auf seiner Website ein Beispiel[2] mit geringer Einstiegshürde und »sichtbaren« Ergebnissen:

```
public class MyUI extends UI {
   protected void init(VaadinRequest request) {
       TextField name = new TextField("Name");
       Button greetButton = new Button("Greet");
       greetButton.addClickListener(
           e -> Notification.show("Hi " + name.getValue())
       );
       setContent(new VerticalLayout(name, greetButton));
   }
}
```

Das Beispiel zeigt die Verwendung von Widgets – eine Besonderheit für Webframeworks. Dieses Beispiel hat den Vorteil, dass mit nur etwa 10 Zeilen Code ein erstes sichtbares Ergebnis entsteht. Das Beispiel kann man für weitere Experimente nutzen, um das Framework auszuprobieren.

2. *https://vaadin.com/introduction#how-works*

2.2.5 Lesbaren Code fördernd

APIs haben enormen Einfluss auf die Lesbarkeit des Clientcodes. Schauen wir uns dazu folgendes Beispiel an:

```
assertTrue(car.getExtras().contains(airconditioning));
assertEquals(2, car.getExtras().size());
```

Das Beispiel ist ein Auszug aus einem Unit-Test. Die beiden Assertions prüfen, ob das Fahrzeug car eine Klimaanlage und insgesamt zwei Extras hat. Alternativ könnte der Unit-Test auch mit dem FEST-Assert-Framework geschrieben werden:

```
assertThat(car.getExtras())
   .hasSize(2)
   .contains(airconditioning);
```

Dank des Fluent Interface, dessen Methodenketten zur Validierung des Testergebnisses stärker an eine natürliche Sprache angelehnt sind, ist der Code des zweiten Beispiels etwas verständlicher. Ein Fluent Interface ist eine Domain Specific Language (DSL), die durch die Anpassung an die Anforderungen ihrer Domäne viel ausdrucksstärker als eine universelle Programmiersprache ist. In Abschnitt 6.1 finden Sie weitere Informationen zu diesem Thema.

Bessere Lesbarkeit und Wartbarkeit von Unit-Tests waren die Entwurfsziele des FEST-Assert-Frameworks. In diesem Zusammenhang könnte man noch viele andere Bibliotheken mit gleichem Zweck nennen: Das Spock Framework beispielsweise bietet eine kleine DSL zur übersichtlichen Strukturierung von Tests.

Ein Beispiel aus einem ganz anderen Aufgabengebiet ist die JPA Criteria API. Diese API dient zur Konstruktion von typsicheren Datenbankabfragen. Mit dem folgenden Java-Code wird eine Query gebaut und ausgeführt, um alle Order-Objekte mit mehr als einer Position zu selektieren:

```
EntityManager em = ...;
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Order> cq = builder
   .createQuery(Order.class);
Root<Order> order = cq.from(Order.class);
order.join(Order_.positions);
cq.groupBy(order.get(Order_.id)).having(
   builder.gt(builder.count(order), 1));
TypedQuery<Order> query = em.createQuery(cq);
List<Order> result = query.getResultList();
```

Übersichtlicher wird die Abfrage mit QueryDSL. Diese Bibliothek bietet ein Fluent Interface, mit dem verständliche Pfadausdrücke formuliert werden können.

```
EntityManager em = ...;
QOrder order = QOrder.order;
JPQLQuery query = new JPAQuery(em);
List<Order> list = query.from(order)
                        .where(order.positions.size().gt(1))
                        .list(order).getResults();
```

Entwickler verbringen mehr Zeit mit dem Lesen als mit dem Schreiben von Quellcode. Daher kann deren Produktivität durch gut lesbaren Quellcode bzw. eine leicht verständliche API verbessert werden. Wie können APIs zu lesbarem Code führen?

- Gute Namenskonventionen sind wichtig, denn sie unterstützen das Lesen und Erfassen des Quellcodes. Gut lesbarer Code enthält auch weniger Fehler, denn Fehler fallen dann schneller auf.
- Die zuvor beschriebenen Eigenschaften Konsistenz und intuitive Verständlichkeit haben ebenfalls einen großen Einfluss auf die Lesbarkeit des Clientcodes.
- Auch ein einheitliches Abstraktionsniveau verbessert die Lesbarkeit von Code. Das bedeutet, dass eine API beispielsweise nicht Persistenzfunktionen mit Geschäftslogik mischen sollte. Das sind Aufgaben unterschiedlicher Abstraktionsniveaus. Wenn diese vermischt werden, entsteht unnötig komplexer Clientcode. Die gewählten Abstraktionen der API sollten passend für die zukünftigen Benutzer ausgewählt werden.
- APIs sollten Hilfsmethoden bieten, sodass der Clientcode kurz und verständlich bleibt. Ein Client sollte nichts tun müssen, was ihm die API abnehmen kann.

2.2.6 Schwer falsch zu benutzen

Eine API sollte nicht nur einfach zu benutzen, sie sollte sogar schwer falsch zu benutzen sein. Daher sollte man nicht offensichtliche Seiteneffekte vermeiden und Fehler zeitnah mit hilfreichen Fehlermeldungen anzeigen. Benutzer sollten nicht gezwungen sein, die Methoden einer API in einer fest definierten Reihenfolge aufzurufen.

Unerwartetes Verhalten

Die ursprüngliche Datums- und Zeit-API von Java sieht auf den ersten Blick einfach und intuitiv aus. Doch schon bei einfachen Beispielen stolpert man über ein Verhalten, das man vermutlich nicht erwartet. Was das bedeutet, wollen wir uns an einem Beispiel anschauen:

Die ursprüngliche Datums- und Zeit-API von Java lädt geradezu dazu ein, Fehler zu machen. Den 20. Januar 1983 würde man vermutlich so definieren wollen:

```
Date date = new Date(1983, 1, 20);
```

Leider enthält diese Codezeile gleich zwei Fehler. Denn die Zeitrechnung dieser API beginnt unerwarteterweise im Jahre 1900. Außerdem sind die Monate beginnend mit 0 durchnummeriert. Die Tage werden beginnend mit 1 angegeben. Deswegen muss der 20. Januar 1983 folgendermaßen erzeugt werden:

```
int year = 1983 – 1900;
int month = 1 - 1;
Date date = new Date(year, month, 20);
```

Im nächsten Schritt geben wir zusätzlich noch die Uhrzeit 10:17 mit der Zeitzone von Bukarest an. Die Uhrzeit soll schließlich in einen formatierten String umgewandelt werden. Weil die Klasse Date keine Zeitzonen unterstützt, müssen wir ein Calendar-Objekt erzeugen. Die erwartete Ausgabe ist »20.01.1983 10:17 +0200«.

```
Date date = new Date(year, month, 20, 10, 17);
TimeZone zone = TimeZone.getInstance("Europe/Bucharest");
Calendar cal = new GregorianCalendar(date, zone);
DateFormat fm = new SimpleDateFormat("dd.MM.yyyy HH:mm Z");
String str = fm.format(cal);
```

Auch hier verstecken sich mehrere Fehler: Der Konstruktor der Klasse GregorianCalendar akzeptiert eine Zeitzone, aber kein Date-Objekt. Der Calendar kann nicht von SimpleDateFormat formatiert werden. Auch SimpleDateFormat muss die Zeitzone übergeben werden. Durch Angabe der Zeitzone wird die Uhrzeit verändert. Der korrigierte Clientcode sieht so aus:

```
int year = 1983 - 1900;
int month = 1 - 1;
// weil 1 Stunde Zeitunterschied zwischen Berlin und Bukarest
int hour = 10 - 1;
Date date = new Date(year, month, 20, hour, 17);
TimeZone zone = TimeZone.getInstance("Europe/Bucharest");
Calendar cal = new GregorianCalendar(zone);
cal.setTime(date);
DateFormat fm = new SimpleDateFormat("dd.MM.yyyy HH:mm Z");
fm.setTimeZone(zone);
Date calDate = cal.getTime();
String str = fm.format(calDate);
```

Aufgrund dieser Fallstricke entstand in der Java-Community die Bibliothek Joda-Time. Der Clientcode könnte folgendermaßen aussehen:

```
DateTime dt = new DateTime(1983, 1, 20, 10, 17,
   DateTimeZone.forID("Europe/Bucharest"));
DateTimeFormatter formatter
   = DateTimeFormat.forPattern("dd.MM.yyyy HH:mm Z");
String str = dt.toString(formatter);
```

Ein anderes nicht unbedingt intuitives Feature ist die Möglichkeit, mehr als 60 Sekunden, mehr als 24 Stunden usw. bei der Erzeugung eines Date-Objektes anzugeben. Statt einer Fehlermeldung wird der Überhang korrekt berechnet. Durch eine Angabe von beispielsweise 25 Stunden wird der nächste Tag 1 Uhr ausgewählt. Dieses Verhalten ist nicht offensichtlich und könnte deswegen zu Fehlern führen.

2.2.7 Minimal

Im Zweifel weglassen!

Eine API sollte prinzipiell so klein wie möglich sein, weil einmal hinzugefügte Elemente nachträglich nicht mehr entfernt werden können. Außerdem sind größere APIs auch komplexer. Dies hat Auswirkungen auf Verständlichkeit und Wartbarkeit der API. Ein ganz anderer Punkt ist der Implementierungsaufwand: Je größer die API, desto aufwendiger ihre Implementierung. Deswegen sollten beispielsweise zusätzliche Hilfsmethoden nur mit Bedacht hinzugefügt werden. Andererseits können Hilfsmethoden sehr nützlich sein. Überhaupt sollte ein Client nichts tun müssen, was eine API übernehmen kann.

Daher braucht man einen Kompromiss, wie ihn die Entwickler der Java-Collection-API gefunden haben: Mit den Methoden addAll und removeAll im Interface java.util.List können mit einem Aufruf mehrere Objekte zu einer Liste hinzugefügt bzw. entfernt werden. Diese Methoden sind optional, weil man Objekte auch einzeln mit add und remove hinzufügen bzw. entfernen kann. Trotzdem ist das Vorhandensein dieser Hilfsmethoden im Interface java.util.List nachvollziehbar und akzeptabel. Diese Hilfsmethoden werden sehr häufig verwendet und passen gut zum Rest des Interface. Andere Hilfsmethoden wie beispielsweise removeAllEven oder removeAllOdd, die alle Objekte mit gerader bzw. ungerader Positionsnummer aus einer Liste entfernen, wären für nur wenige spezielle Anwendungsfälle hilfreich und gehören deswegen nicht in die API.

Die Ruby-Standardbibliothek hat diverse Methoden mehrfach, weil man so im Clientcode besser ausdrücken kann, was man tut. Die Anzahl der Elemente eines Arrays kann z. B. mit length, count und size abgefragt werden. Das muss man nicht ermöglichen, aber es ist ein guter Stil, wenn man ihn konsistent anwendet.

Weniger ist manchmal mehr

Schweizer Messer sind bekannt für ihre zahlreichen Werkzeuge. Neben einer Klinge bieten sie z. B. eine Holzsäge, einen Korkenzieher, eine Schere, eine Metallfeile oder eine Pinzette. Manche dieser Werkzeuge werden kaum oder vielleicht nie benutzt. Ein gewöhnlicher Schraubenzieher mit einem vergleichsweise einfachen Design ist ebenfalls vielseitig einsetzbar: Man kann beispielsweise eine Farbdose mit ihm öffnen, falls der Deckel klemmt. Man kann mit ihm die Farbe umrühren, ein Loch in etwas machen, etwas hinter dem Schrank hervorholen, das man mit der Hand nicht erreichen kann, und man kann sogar Schrauben festdrehen.

Auch eine kleine, einfache API kann vielseitig einsetzbar sein. Es muss nicht für jeden Sonderfall eine spezielle Funktion, die am Ende kaum jemand nutzen wird, eingebaut werden. Dennoch sind Schweizer Messer sehr nützlich.

2.2.8 Stabil

Stabilität ist eine entscheidende Eigenschaft von APIs. Stellen Sie sich vor, Sie entwickeln einen Tarifrechner, der sich als großer Erfolg erweist und in zahlreiche Kundensysteme integriert werden soll. Diese Integrationen werden von verschiedenen Teams durchgeführt und sind oft kostspielig, da Altsysteme nur mit erheblichem Aufwand angepasst werden können. Nun gibt es eine neue Anforderung aus der Fachabteilung: Die komplexen Berechnungsregeln des Tarifrechners müssen erweitert werden.

Sollte sich die API oder das Verhalten der Schnittstelle dabei ändern, könnten Integrationsprobleme mit den Altsystemen entstehen. Daher muss jede Änderung sorgfältig daraufhin geprüft werden, ob sie negative Auswirkungen auf bestehende Nutzer hat. Außerdem sollte überlegt werden, wie diese Änderungen angemessen kommuniziert werden können.

In Kapitel 7 werden wir genauer untersuchen, welche Änderungen rückwärtskompatibel sind. Wenn Änderungen nicht kompatibel sind, muss unter Umständen eine neue Version der API bereitgestellt werden.

Auch bei der Einführung einer neuen API-Version bleibt Stabilität von großer Bedeutung. Ziel sollte es sein, die Migration für bestehende Clients so einfach wie möglich zu gestalten.

2.2.9 Einfach erweiterbar

Eine weitere Eigenschaft von APIs ist Änderbarkeit, ein zentrales Qualitätsmerkmal für Softwareprodukte. Man versteht darunter den erforderlichen Aufwand zur Durchführung vorgegebener Änderungen für Korrekturen, Verbesserungen oder Anpassungen an neue Anforderungen. Diese Eigenschaft ist kein Widerspruch zur zuvor genannten Stabilität, denn gemeint ist Folgendes:

- Bei der Erweiterung einer API sollte der Änderungsaufwand für existierende Clients berücksichtigt werden. Im nächsten Abschnitt wird aus diesem Grund die Metrik Konnaszenz vorgestellt.
- Im Idealfall ist die veränderte API kompatibel und der Clientcode muss überhaupt nicht angepasst werden.

Eine Java-API kann beispielsweise durch Vererbung erweitert werden:

- API-Benutzer können durch Vererbung das Verhalten eines Frameworks anpassen oder ändern.
- API-Entwickler können eine neue Subklasse hinzufügen, um auf kompatible Art und Weise neue Funktionen umzusetzen. Die neue Subklasse wird womöglich in einer Factory-Methode erzeugt, sodass API-Benutzer dies nicht bemerken.

Auch für Web-APIs ist Änderbarkeit ein wichtiges Qualitätsmerkmal. Flexible Datenformate wie XML und JSON können genutzt werden, um kompatible Erweiterungen umzusetzen.

2.3 Konnaszenz

Für leicht änderbaren Code ist noch keine Zauberformel erfunden worden, aber mit Konnaszenz (engl. Connascence) steht ein gutes Modell zur Verfügung, mit dem wir zumindest über die Änderbarkeit von APIs sprechen können.

Was ist Konnaszenz?

Konnaszenz berücksichtigt verschiedene Typen von Kopplung sowie deren Häufigkeit und Lokalität für Aussagen über die Änderbarkeit von Code (The connascence.io website, 2018). Zwei Komponenten A und B gelten im Sinne von Konnaszenz als gekoppelt, falls eine Änderung an A eine Änderung an B erfordert, um die Korrektheit des Gesamtsystems zu gewährleisten. Eine starke Form von Konnaszenz zwischen A und B ist nur akzeptabel, wenn beide Komponenten eng beieinander liegen, weil sie zum Beispiel Teil derselben Codebasis sind. Umgekehrt ist eine starke

Form von Konnaszenz zwischen Komponenten unterschiedlicher Systeme inakzeptabel. Es gilt: Je stärker die Konnaszenz zwischen A und B, desto schwieriger und aufwendiger sind deren Änderungen. Konnaszenz wird mit den Dimensionen Stärke, Häufigkeit und Lokalität beschrieben. Dies kann beim API-Design genutzt werden, um die Verbindung zwischen API und Client zu analysieren:

Stärke

- Stärkere Formen von Konnaszenz sind schwerer zu finden oder zu ändern als leichtere Formen. Beispielsweise sind Namensänderungen leicht zu entdecken und durchzuführen. Im Vergleich dazu ist die Anpassung eines Algorithmus, der zwischen API und Client abgestimmt werden muss, schwierig.

Häufigkeit

- Die Änderung einer API, die sehr viele Clients hat, ist höchstwahrscheinlich eine größere Herausforderung als die Änderung einer API mit wenigen Clients. Die Änderbarkeit von öffentlichen APIs mit vielen Clients ist daher stark eingeschränkt.

Lokalität

- Je weiter API und Client entfernt sind, desto schwieriger sind Änderungen. Wenn API und Client zur selben Codebasis gehören, ist die Änderung vergleichsweise einfach. Änderungen innerhalb des Zuständigkeitsbereichs eines Entwicklungsteams sind ebenfalls akzeptabel. Aber Änderungen über Organisationsgrenzen hinweg sind deutlich komplexer, weil die Organisationen wahrscheinlich verschiedene Ziele verfolgen, unterschiedliche Releasezyklen haben und die Kommunikation insgesamt schwieriger oder aufwendiger ist als innerhalb eines Teams.

Statische Konnaszenz

Es gibt verschiedene Formen von Konnaszenz, die man in statisch und dynamisch einteilen kann. Statische Formen kann man durch Lesen des Codes finden und analysieren. Die folgende Liste ist nach aufsteigender Stärke sortiert:

Namenskonnaszenz

- Die schwächste Form statischer Konnaszenz ist Namenskonnaszenz. In diesem Fall müssen sich mehrere Komponenten auf den Namen eines Elements einigen. Falls zum Beispiel ein Name im JSON-Payload einer API geändert wird, müssen auch die von diesem Format abhängigen Clients angepasst werden.

Typkonnaszenz

- Mehrere Komponenten müssen sich auf den Typ eines Elements einigen. Die Änderung eines Integers im JSON-Format einer API in einen String ist eine inkompatible Änderung mit mittelschwacher Konnaszenz.

Bedeutungskonnaszenz

- Mehrere Komponenten müssen sich auf die Bedeutung bestimmter Werte einigen. Wenn eine API in ein Feld preis statt des Nettopreises nun den Bruttopreis schreibt, müssen die Clients angepasst werden.

Positionskonnaszenz
- Mehrere Komponenten müssen sich auf die Reihenfolge bestimmter Werte einigen. Die Reihenfolge von Parametern für einen API-Aufruf ist ein typisches Beispiel für Positionskonnaszenz.

Algorithmuskonnaszenz
- Bei der stärksten Form statischer Konnaszenz müssen sich mehrere Komponenten auf einen bestimmten Algorithmus einigen. Verbindungen zwischen API und Clients dieser Stärke können nur mit großem Aufwand geändert werden, daher sollte man beim Entwurf einer API auf Algorithmuskonnaszenz achten und versuchen, diese zu minimieren. Denken Sie beispielsweise an IBANs, die einem bestimmten Format folgen. Der Algorithmus zur Validierung von IBANs wird von unzähligen Systemen genutzt. Eine Änderung dieses Algorithmus wäre äußerst schwierig.

Dynamische Konnaszenz

Wie zuvor erwähnt, existieren neben statischen auch dynamische Formen der Konnaszenz, die im Folgenden näher erläutert werden:

Ausführungskonnaszenz
- Die Methoden einer API können nur in spezieller Reihenfolge aufgerufen werden. Häufig gibt es einen offensichtlichen fachlich-logischen Grund, warum API-Methoden nicht in beliebiger Reihenfolge benutzt werden können. Nehmen wir zum Beispiel einen Webservice einer Shopping-Anwendung zum Verwalten von elektronischen Einkaufswagen. Man kann beispielsweise nicht mit einer Operation CartAdd einen Artikel in einen Einkaufskorb legen, wenn nicht zuvor mit einer Operation CartCreate ein entsprechendes Objekt (Einkaufskorb) mit dem Webservice erzeugt wurde.

Zeitliche Konnaszenz
- Diese Form bezieht sich auf die zeitliche Koordinierung zwischen API und Client. Fachliche Transaktionen werden beispielsweise für den Check-out-Prozess der Shopping-Anwendung eingesetzt. Nach Ablauf einer Frist wird die Transaktion automatisch vom Webserver gelöscht, es sei denn, die Transaktion wird vorher vom Client erfolgreich beendet oder abgebrochen.

Wertekonnaszenz
- Diese Form von Konnaszenz liegt vor, wenn sich mehrere Werte zusammen ändern müssen, wenn beispielsweise Client und Server sich auf bestimmte Zahlen einigen, um den Zustand von Bestellungen anzugeben. Eine Bestellung in Bearbeitung hat dann zum Beispiel den Wert 2 und eine abgeschlossene Bestellung den Wert 3.

Identitätskonnaszenz
- In diesem Fall müssen mehrere Komponenten dieselbe Entität referenzieren. Angenommen, die zuvor erwähnte Shopping-Anwendung besteht aus mehreren getrennten Webservices für die Verwaltung der elektronischen Einkaufswagen, für den Check-out-Prozess und für die Artikelsuche. Identitätskonnaszenz liegt vor, wenn die Webservices für Check-out und Artikelsuche denselben Einkaufswagen referenzieren müssen.

Konnaszenz gibt Ihnen das notwendige Vokabular, um die Änderbarkeit von APIs zu untersuchen und die vielfältige Kopplung zwischen Client und API zu benennen.

2.4 Zusammenfassung

In diesem Kapitel haben Sie die Qualitätsmerkmale bzw. Qualitätsziele kennengelernt. Diese sind hier zusammengefasst:

- APIs müssen vollständig und korrekt sein.
- APIs sollten konsistent, intuitiv verständlich, dokumentiert, minimal, stabil, erweiterbar und leicht zu lernen sein. Sie sollten es Benutzern leicht machen, lesbaren Code zu schreiben. Es sollte schwer sein, sie falsch zu benutzen.
- Die Änderbarkeit von APIs kann mit der Metrik Konnaszenz systematisch analysiert werden.

Im folgenden Kapitel werden Sie erfahren, wie APIs auf Basis von Use Cases und Beispielen entsprechend zuvor identifizierter Anforderungen iterativ mit Feedbackschleifen entworfen werden können.

Inhaltsübersicht

Inhaltsverzeichnis