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 5.7 und 9.4. *Zuverlässigkeit*

Kai Spichale: API-Design (ISBN Print: 978-3-86490-387-8)

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 folgenen 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 Namenskonventio-

nen sind in Ordnung. Es ist eine Frage des persönlichen Geschmacks, welche man bevorzugt.

Was ist das Problem? Das Problem ist, 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!
```

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

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

Setter- und With-Mehoden

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 Joda-Time genutzt.

Methoden der Java-Collections

Ein anderes anschauliches Beispiel sind die Collections der Java-Standardbibliothek. Die starken Begriffe add, contains und remove wurden hier etabliert und da, wo es passt, wiederverwendet. Man findet diese Methoden einheitlich in den Interfaces List und Set. Für Map wurde leider ein anderer Name verwendet. Die Methode zum Hinzufügen von Elementen heißt dort put. Zugegeben, diese Methode hat andere Parameter und funktioniert nicht wie add von List, aber auch zwischen List und Set gibt es Unterschiede. Ein einheitliches add in allen drei Interfaces wäre vermutlich besser 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 Tabelle zeigt eine gewisse Symmetrie. Denn es gibt Methodenpaare wie add und remove, addAll und removeAll etc. Die Methode removeAll scheint bei Map zu fehlen, denn diese Methode wäre das Gegenstück zu putAll. Map bietet außerdem die Methode entrySet und keySet, aber nicht die Methode valueSet. Diese Methode heißt korrekterweise values, weil die Mengeneigenschaften in diesem Fall nicht erfüllt werden kann.

Es ist wichtig, starke Begriffe in einer API zu etablieren und diese einheitlich wiederzuverwenden. Beispielsweise sollten Sie nicht Synonyme wie »delete« und »remove« innerhalb einer API beliebig mischen. Entscheiden Sie sich für einen Begriff und bleiben Sie dann dabei. Verwenden Sie Begriffe, die den Benutzern der API geläufig sind. Das Wort »erase« wäre zum Beispiel zu ungewöhnlich. Ein Java-

Entwickler würde vermutlich für Dateioperationen nach »create« und »delete« als Erstes suchen. Bei Persistenzoperationen hingegen nach »insert« und »remove«.

Eine API ist intuitiv verständlich, wenn Entwickler den Clientcode der API ohne die Dokumentation lesen können. Das kann nur durch Vorwissen und sprechende Bezeichner funktionieren. Daher sollten Sie gezielt versuchen, Begriffe aus bekannten APIs wiederzuverwenden.

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:

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

```
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.

2.2.5 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 Famework 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 Daten-

bankabfragen. 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. einer leicht verständlichen 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. Nichtsdestotrotz sind Schweizer Messer sehr nützlich.

2.2.8 Stabil

Stabilität ist eine wichtige Eigenschaft von APIs. Angenommen Sie entwickeln einen Tarifrechner. Ihr Produkt wird ein großer Erfolg und soll in mehrere Kundensysteme integriert werden. Die Integrationen werden von unterschiedlichen Teams durchgeführt und sind relativ teuer, weil Altsysteme nur mit großem Aufwand angepasst werden können. Dann gibt es eine neue Anforderung aus der Fachabteilung und die komplexen Berechnungsregeln des Tarifrechners müssen erweitert werden. Falls sich nun die Schnittstelle oder das bisherige Verhalten dieser Schnittstelle ändern würde, gäbe es Probleme bei der Integration in die Altsysteme. Deswegen muss bei jeder Änderung geprüft werden, ob diese negative Auswirkungen auf bestehende Benutzer hat und wie diese gegebenenfalls kommuniziert werden können. In Kapitel 7 werden wir uns anschauen, welche Änderungen kompatibel sind. Falls

Änderungen nicht kompatibel sind, ist gegebenenfalls eine neue Version zu nutzen. Stabilität ist auch bei Einführung einer neuen Version wichtig, wenn Sie die Migration für existierende Clients möglichst einfach machen wollen.

2.2.9 Einfach erweiterbar

Eine weitere Eigenschaft von APIs ist Erweiterbarkeit. 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 Idealfall ist die veränderte API kompatibel und der Clientcode muss überhaupt nicht angepasst werden.

Eine 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 Erweiterbarkeit ein wichtiges Qualitätsmerkmal. Flexible Datenformate wie XML und JSON können genutzt werden, um kompatible Erweiterungen umzusetzen.

2.3 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.

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.

Kai Spichale: API-Design (ISBN Print: 978-3-86490-387-8)