

# API-Design

v.1.0.0

## Geschichte

### Subroutinen

- 1948: Hermann Goldstine und John von Neumann beschreiben die Idee von Subroutinen
- 1951: Maurice V. Wilkes und sein Team (u.a. David Wheeler) bauten den EDSAC-Röhrencomputer
  - inklusives detailliertes Schema zum Einsatz von Subroutinen



*EDSAC Röhrencomputer*

- 1952: Wheeler veröffentlicht ein 2-Seiten-Paper mit grundlegenden Konzepten für Subroutinen
  - Subroutinen / Subroutinen und Bibliotheken
  - Bedeutung von Dokumentation für Subroutinen-Bibliotheken
  - Geheimnisprinzip
  - Trade-off zwischen Generalität und Performance
  - Funktionen höherer Ordnung
  - Debugger
  - Routinen zur Interpretierung von Pseudocode
- 1967: Wilkes bekommt den Turing Award für die Idee von Subroutinen-Bibliotheken
- 1968: der Begriff *Application Programming Interface* wird geprägt

## NOTE

Wheeler unterschied noch nicht zwischen Implementierung und API, da es nur eine Maschinenarchitektur gab (und somit keine anderen Implementierungen). Erst mit neuer Hardware und dem Wunsch, vorhandene Programme zu Portieren, wurde zwischen API und Implementierung unterschieden.

# Web-APIs

- 2000, Februar: **Salesforce.com** startet offiziell seinen XML-basierten Dienst
  - XML-API ermöglicht den Informationsaustausch programmübergreifend
- 2000, November: Live-Schaltung der **eBay**-API und des eBay Developer Programs
  - das Parsen der Webseite wurde überflüssig und eine gute Integration in Dritthersteller-Applikationen möglich
- 2004: das Webportal **Flickr** geht online
  - durch die RESTful API wurde das Portal schnell populär für Blogger und Nutzer sozialer Medien
- 2006: **Facebook**, **Twitter** und **Google Maps** geben eine offizielle REST-API frei
  - im gleichen Jahr gründeten sich die ersten API-Serviceprovider
- ~2006: *Public Cloud Computing Platform as a Service* (PaaS) wird populär und findet Verbreitung über Web-APIs
- aktuell:
  - *Internet of Things* (IoT) gewinnt an Bedeutung und verlässt sich stark auf APIs
  - Trend von PaaS, Software as a Service (SaaS) und Infrastructure as a Service (IaaS) setzt sich fort

## NOTE

- API-Serviceprovider: Bereitstellung von Infrastruktur zur Entwicklung, Veröffentlichung und Verwaltung von APIs



*IoT und Cloud*

## Definition API

"Eine API spezifiziert die Operationen sowie die Ein- und Ausgabe einer Softwarekomponente. Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass die Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen." — Joshua Bloch

"Eine Programmierschnittstelle, genauer Schnittstelle zur Anwendungsprogrammierung, [...] ist ein Programmteil, der von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird." — Wikipedia, 2018

**NOTE** | Eine API ist immer ein Vertrag und keine Implementierung.

## Programmiersprachen- und Remote-APIs

- Programmiersprachen-APIs
  - z.B. von Bibliotheken oder Frameworks
  - Service Provider Interfaces (SPIs) werden vom Nutzer implementiert / erweitert (Erweiterungspunkte)
- Remote-APIs
  - z.B. RESTful HTTP, Messaging-APIs oder Remote Procedure Calls (RPCs)

# Abgrenzung

API != (technisches) Protokoll API > (technisches) Protokoll

## NOTE

Beispiel: HTTP ist ein Protokoll und stellt alleine keine API dar; erst eine Menge von HTTP-Requests und -Responses inklusive der Definition von Nachrichteninhalten.

# Vorteile

## Aus technischer Sicht

- Stabilität durch lose Kopplung
  - Änderungen in der Implementierung beeinflussen **nicht** den Nutzer
- Portabilität
  - z.B. die JRE bietet für Java-Programme eine einheitliche API für verschiedene Betriebssysteme
- Reduktion von Komplexität
  - Komplexität der Implementierung wird versteckt (Geheimnisprinzip)
- Modularisierung
  - ein API-Modul erfüllt eine Aufgabe
  - unterschiedliche Teams können besser (zusammen) arbeiten
- Wiederverwendbarkeit und Integration
  - Zugriff auf Funktionalität einer (fremden) Softwarekomponente wird ermöglicht

## NOTE

Um die Vorteile zu erreichen, muss die API gut entworfen sein. Schlechte APIs machen die Vorteile zu nichts und führen zu erhöhtem Entwicklungsaufwand seitens der Nutzer bis hin zur Ablehnung der Integration.

## Aus wirtschaftlicher Sicht

- (erleichterte) Client-Entwicklung
  - unterschiedliche Endgeräte greifen auf die gleiche API zu (z.B. Facebook)
- Benutzerakzeptanz erhöhen
  - gute APIs führen zu mehr Integration und damit zu mehr Verbreitung
  - z.B. konnte Twitter mehr Nutzer allein durch die API-Veröffentlichung gewinnen
- neue Geschäftszweige
  - z.B. wurde die *BestBuy*-API genutzt, um Preisvergleich, Preiswecker etc. zu implementieren
- Integration mit Partnern

- Vernetzung mit Partnern wird ermöglicht (z.B. Lieferdienste)
- Integration unternehmensintern
  - klarere Aufgabenabgrenzung
  - Wiederverwendbarkeit in anderen Teams / Abteilungen / Programmen

## Nachteile

- fehlende Interoperabilität
  - z.B. kann eine Java-API nicht von einer C#-Applikation konsumiert werden
- Änderbarkeit
  - ausgerollte APIs können erstmal nicht geändert werden

### NOTE

Das Problem der Interoperabilität lässt sich mit Remote-APIs lösen - es ist jedoch erstmal ein Problem.

## Qualitätsmerkmale

- Benutzbarkeit
  - für andere leicht verständlich
  - **zentrales Ziel**
- Effizienz
  - z.B. bei mobilen Applikationen den Akku-Verbrauch im Blick haben
  - Skalierbarkeit - kann die API mit Nutzerzuwachs mithalten?
- Zuverlässigkeit
  - Fehlerbehandlung
- vollständig und korrekt

### NOTE

Die Qualitätsmerkmale basieren auf ISO 9126.

## Benutzbarkeit

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

- stabil
- einfach erweiterbar

## Konsistenz

Beispiel:

str_repeat	strcmp
str_split	strlen
str_word_count	strrev

- beide Namenskonventionen sind ok
- Problem: alle Funktionen sind Teil der gleichen (PHP-)API

## Konzeptionelle Integrität

- ein Architekt (oder eine kleine Gruppe von Architekten) entscheidet über Konzepte
- alles außerhalb der Konzepte wird nicht umgesetzt bzw. entsprechend angepasst
- "Konzeptionelle Geschlossenheit ist der Dreh- und Angelpunkt für die Qualität eines Produkts [...]" — Fred Brooks

### NOTE

Fred Brooks ist der Autor des Buchs *The Mythical Man-Month: Essays on Software Engineering*.

## weiteres Gegenbeispiel von Java:

- `javax.swing.AbstractButton.setText()`
- `java.awt.Button.setLabel()`

### NOTE

Die konzeptionelle Integrität und damit die Konsistenz ist hier nicht gegeben. Die spätere Funktionalität hätte sich namentlich an der frühere orientieren müssen oder die früher ebenfalls abändern.

## Intuitiv

- einheitliche Namenskonvention
  - innerhalb der API (Konsistenz!)
  - innerhalb der Programmiersprache
  - innerhalb etablierter Konzepte
  - gleiche Dingen haben den gleichen Namen
  - unterschiedliche Dinge haben unterschiedliche Namen

## NOTE

Viele Konzepte haben mehrere Namen (z.B. Observer / Listener). Ein Konzept sollte innerhalb einer API immer genau einen Namen haben. Man könnte aber z.B. die unterschiedlichen Namen für unterschiedliche Subtypen benutzen - hier natürlich auch konsistent.

### Beispiel von Ruby:

- Methoden mit '!' am Ende, verändern das Objekt
- Methoden ohne '!' am Ende, erzeugen eine neue Instanz und das ursprüngliche Objekt ändert sich nicht
- `my_string.capitalize`
  - `my_string.capitalize!`
- `my_string.reverse`
  - `my_string.reverse!`
- Wie heißen die Methoden für `downcase`?
  - `my_string.downcase` und `my_string.downcase!`

### Java Konventionen:

- `set` Methoden verändern das Objekt
- `get` Methoden geben einen Wert zurück
- `add` fügt etwas hinzu
- `put` fügt etwas hinzu oder überschreibt vorhandenes

List	Set	Map
<ul style="list-style-type: none"><li>• geordnete Menge</li><li>• Duplikate sind erlaubt</li><li>• <code>null</code> ist erlaubt</li></ul>	<ul style="list-style-type: none"><li>• ungeordnete Menge</li><li>• keine Duplikate</li><li>• höchstens ein <code>null</code> Element</li></ul>	<ul style="list-style-type: none"><li>• Schlüssel → Wert Zuordnung</li><li>• keine Schlüssel-Duplikate erlaubt</li><li>• pro Schlüssel höchstens 1 Wert</li></ul>

<code>java.util.List</code>	<code>java.util.Set</code>	<code>java.util.Map</code>
<code>add / addAll</code>	<code>add / addAll</code>	<code>put / putAll</code>
<code>remove</code>	<code>remove</code>	<code>remove</code>
<code>removeAll</code>	<code>removeAll</code>	<code>clear</code>

## NOTE

- `put` würde anstatt `add` bei `Set` mehr Sinn machen, da ein `Set` genauso wie `Map` keine Duplikate erlaubt
- `removeAll` anstatt `clear` sollte die Methode bei `Map` heißen

# Dokumentation

- *gute* Dokumentation ist unverzichtbar
  - lieber keine Dokumentation als falsche Dokumentation
  - lieber wenig Dokumentation als keine Dokumentation
- sollte einfache Beispiele zur korrekten Nutzung enthalten
  - Entwickler sollten nur wenig von einem Beispiel anpassen müssen
- Beispiel Spring: die Dokumentation (mit zahlreichen Beispielen) hat vermutlich stark zur Akzeptanz beigetragen

## NOTE

Dokumentationsmöglichkeiten:

- im Code (z.B. JavaDoc)
- eine extra Readme Datei (z.B. github)
- eine Wiki / eine Webseite
- ...

# Einprägsam und leicht zu lernen

- Konsistenz, intuitive Nutzung und Dokumentation erleichtern das Erlernen
- Boiler-Plate-Code auf ein Minimum reduzieren
- mit wenig Code *sichtbare* Ergebnisse ermöglichen

## NOTE

Boiler plate: Ursprung des Begriffs in der heutigen Nutzung in den Printmedien. Vorgefertigter Text (d.h. vorgesetzter Text - im Zeitalter vor digitalen Printmedien), der auf Platten kam, wurde aufgrund des ähnlichen Aussehens mit Kesselplatten gleichgesetzt. Da er meist wenig Inhalt enthielt und nur als *Füller* benutzt wurde, wurde *boilerplate text* schnell ein Synonym für unnötiger, unnützer oder inhaltsloser Text.

Beispiel von Vaadin (<https://vaadin.com/flow>):

```
public class MyUi extends VerticalLayout {
    public MyUi(@Autowired HelloService service) {
        TextField name = new TextField("Name");
        Button button = new Button("Say hello", click ->
            service.sayHello(name.getValue()));
        add(name, button);
    }
}
```



## NOTE

Mit nur wenigen Zeilen Code wird ein Fenster mit Textfeld und Button erzeugt, das zudem einfache Funktionalität beim Klick auf den Button enthält.

# Lesbaren Code fördernd

- APIs beeinflussen maßgeblich die Lesbarkeit des Clientcodes
- mehr Code wird gelesen als geschrieben
- gute Namen und Konsistenz
- einheitliches Abstraktionsniveau
  - z.B. Trennung von Geschäftslogik und Persistenz
- API muss möglichst viel abnehmen
  - `MyAPI.magic()`
  - Clientcode bleibt kürzer

## Unit-Tests mit JUnit:

```
assertTrue(pizza.extras().contains(Extras.Garlic));
assertEquals(3, pizza.extras().size());
```

## Unit-Tests mit FEST-Assert-Framework:

```
assertThat(pizza.extras())
    .hasSize(3)
    .contains(Extras.Garlic);
```

## JPA Criteria Builder API vs QueryDSL Library

### JPA Criteria Builder

```
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();
```

### QueryDSL

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

## Schwer falsch zu benutzen

- Seiteneffekte vermeiden
- Fehler zeitnah mit hilfreichen Meldungen
- keine temporäre Kopplung

### Ursprüngliche Zeit-API in Java:

```
Date date = new Date(1984, 2, 23)
```

- Zeitrechnung beginnt 1900
- Monate beginnen bei 0
- Tage beginnen bei 1
- obiges Datum wäre: 23.3.2884

#### NOTE

Die API ist inkonsistent (Monate beginnen bei 0, Tage bei 1) und nicht intuitiv (Zeitrechnung ab 1900).

Beides führt zu Fehlern.

### Date in JavaScript

```
var xmas95 = new Date('December 25, 1995 23:15:30');
console.log(xmas95.getYear());
console.log(xmas95.getMonth());
console.log(xmas95.getDay());
```

- 1995
- 11
- 1
  - `getDay()` liefert den Wochentag zurück (0 = Sonntag, 1 = Montag, ...)

# Minimal

- prinzipiell gilt: so klein wie möglich
  - im Zweifel Methoden weglassen
- nachträglich Dinge entfernen ist nahezu unmöglich
  - hinzufügen dagegen einfach
- größere APIs sind aufwendiger zu implementieren und zu warten
- größere APIs sind schwerer zu benutzen
- Kompromiss zwischen Hilfsmethoden und Minimalismus notwendig
- Beispiel: `java.util.List`
  - `removeAll` und `addAll` wurden implementiert
  - mit `remove` und `add` könnte man das Gleiche erreichen
  - `*All`-Methoden werden aber häufig gebraucht

## Schweizer Taschenmesser VS Schraubenzieher

- auch ein Schraubenzieher ist vielseitig
  - Schrauben reindrehen
  - Farbdeckel öffnen
  - verlängerter Arm um Sachen unter dem Schrank vorzuholen
- Schweizer Taschenmesser ist trotzdem nützlich



### NOTE

Auch kleine APIs können vielseitig einsetzbar sein. Große APIs haben u.U. aber auch ihre Daseinsberechtigung.



#### NOTE

Verhältnismäßigkeit sollte aber unbedingt beachtet werden. Und wie schon erwähnt: Im Zweifel Funktionalität weglassen.

## Stabil

- nur Erweiterungen hinzufügen, die mit der ursprünglichen API kompatibel sind
- ggf. eine neue API-Version rausbringen
  - Migrationspfad beschreiben
- Beispiel: API wurde schon in Alt-Systeme integriert, was zeit- und kostenaufwendig ist
  - jede API-Änderung sollte die Integration nicht gefährden

## Einfach erweiterbar

### Bei Erweiterungen der API:

- im Idealfall ist die Änderung kompatibel zu bestehendem Clientcode
- Änderungsaufwand für Clients muss berücksichtigt werden
- z.B. durch (neue) Subklassen, die über eine Factory erzeugt werden

### Zur Erweiterung durch die Clients:

- z.B. durch Vererbungen oder Annotationen das Verhalten des Frameworks anpassen

# Vorgehen beim Entwurf

## Allgemein

- ähnlich wie andere Software auch
- nur Heuristiken vorhanden - es gibt kein deterministisches Vorgehen
- häufig entstehen APIs informell
  - Entwickler macht sich vor seinem PC Gedanken und implementiert dann
  - für kleinere Projekte ok, für größere Projekte **schlecht**

### Allgemeines gutes Vorgehen:

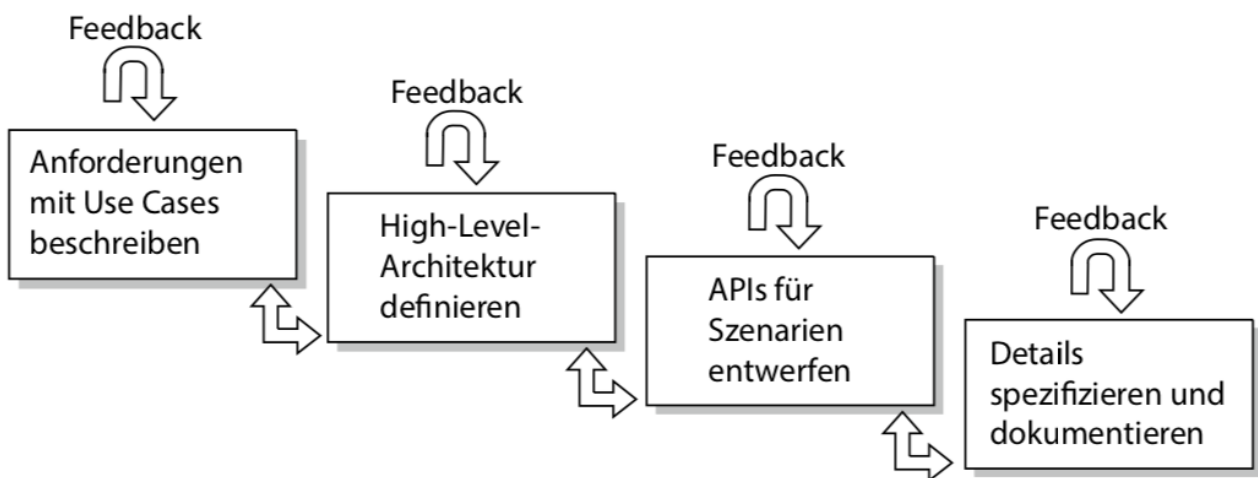


Bild-Quelle: API-Design von Kai Spichale, S. 25

#### NOTE

Wichtig sind die Feedback-Schleifen. Je mehr Feedback - auch negatives - umso besser. Jederzeit kann man auch zum vorherigen Schritt zurück, wenn etwas nicht richtig passt.

## Anforderungen analysieren

- API in das Gesamtprojekt / die Gesamtarchitektur einordnen
  - untergeordnete oder zentrale Rolle?
- Anforderungen VS falsche Annahmen, Missverständnisse und politische Entscheidungen
  - Anforderungen müssen *ausgegraben* werden
- einschränkende Anforderungen kritisch hinterfragen
  - z.B. wenn implizit eine technische Lösung vorgegeben wird
- Anforderungen einfach und verständlich formulieren
  - dabei aber genau sein

- schlecht: *API soll intuitiv benutzbar sein*
- besser: *API soll primär für Java-Entwickler entwickelt werden*
- oft stehen sich Anforderungen entgegen (z.B. Einfachheit - Sicherheit)
  - Abstriche begründet kommunizieren und Feedback der entsprechenden Benutzergruppe einholen

## Use Cases

- etablierte Technik für Benutzeranforderungen
  - Wie reagiert das System auf Benutzeraktionen?
- aus Anforderungen Use Cases ableiten
  - ein Use Case deckt mehrere ähnliche Szenarien ab
- eher ungeeignet für server- oder hardwarelastige Systeme
- mögliche Alternative: Event-Response-Tabellen
  - wie reagiert das System auf bestimmte Ereignisse

### NOTE

Auf Use Cases wird hier im Detail nicht eingegangen, da dies ein eigenes Thema ist.

Event-Response-Tabelle: Event | Trigger | Source | Activity | Response | Destination

## Entwurf

### Erster Schritt:

- kleiner Entwurf
  - Feedback-Einholen einfacher
- Beispiele aus Clientperspektive zeigen
  - z.B. mit Sequenzdiagrammen oder informell am Whiteboard
- eher auf konkreten Code verzichten

### Zweiter Schritt:

- Code-Beispiele, die Use Cases abdecken
  - mit den höher priorisierten UCs anfangen
- verschiedene Konzepte ausprobieren
  - Vor-/Nachteile abwägen
- keine Details festlegen

### Beispiel Audit-Logs:

Szenario 1: Erfolgreiche Benutzeranmeldung loggen

```
AuditLogger logger = new AuditLogger();
AuditEvent event = logger.event()
    .name("user login")
    .status(SUCCESS)
    .user("user1")
    .date(new Date())
    .build();
logger.log(event);
```

## Szenario 2: Fehlgeschlagene Benutzeranmeldung loggen

```
AuditLogger logger = new AuditLogger();
AuditEvent event = logger.event()
    .name("user login")
    .status(FAILURE)
    .user("user1")
    .date(new Date())
    .detail("failure explanation", "wrong password") .detail("user locked", "true")
    .build();
logger.log(event);
```

## Resultierende API:

```
public class AuditLogger {
    public AuditLogger() { }
    public AuditEventBuilder event() { }
    public void log(AuditEvent event) { }
}

public class AuditEventBuilder {
    public AuditEventBuilder name(String name) { }
    public AuditEventBuilder status(String success) { }
    public AuditEventBuilder user(String user) { }
    public AuditEventBuilder date(Date date) { }
    public AuditEventBuilder detail(String key, String value) { }
    public AuditEvent build() { }
}

public class AuditEvent {
    // Getter-Methoden
}
```

## Spezifikation

- **allgemein:** Eigenschaften und Konzepte für die Gesamt-API
- **Module:** für alle Elemente innerhalb eines Moduls, sowie der Zweck des Moduls

- **Klassen/Interfaces:**
  - kurze Beschreibung des Objekts
  - Angaben zu: Thread-Sicherheit, Zustand (Immutability), Serialisierung, Sicherheitseinschränkungen, (Hardware-)Abhängigkeiten, Verweis auf andere Dokumente
- **Felder:** Bedeutung, Wertebereich und `null` erlaubt
- **Methoden und Konstruktoren:**
  - erwartetes Verhalten
  - ändert der Aufruf das Objekt; falls ja, wie
  - Beschreibung der Parameter: Wertebereich, `null` erlaubt, Rückgabewerte (`null` erlaubt), Exceptions
  - evtl. Algorithmus beschreiben

#### NOTE

- Natürlich müssen in erster Linie die öffentlichen Module, Klassen, Felder und Methoden beschrieben werden.
- Öffentliche Elemente markieren, die nicht zur Benutzung gedacht sind (falls z.B. die Sprache nicht ausreichend Einschränkungen nach außen ermöglicht).

## Reviews und Feedback

- Reviews und Feedback sind sehr wichtig (*Betriebsblindheit*)
- Kollegen, Team und API-Nutzer gezielt und regelmäßig um Rückmeldung bitten
- auch negatives Feedback ist hilfreich; je mehr Feedback, umso besser

#### Negativ-Beispiele:

- `creat` anstatt `create`
  - Ken Thompson - Erfinder von Unix
- `referer` anstatt `referrer`
  - HTTP-Header
- `properites` anstatt `properties`
  - Forschungsprojekt mit mehreren Partnern

#### NOTE

Solche Schreibfehler sind nicht nur ärgerlich, sie kosten auch Zeit und Ressourcen. In dem besagten Forschungsprojekt hat es über einen halben Tag gedauert, bis man herausgefunden hat, dass einer der Partner die API falsch implementiert hat (`properites` anstatt `properties`). Dazu kam noch die Zeit, den Fehler zu beheben (durch verteiltes Arbeiten und Absprachen zieht sich so ein Prozess über mehrere Tage).



# Wiederverwendung

- bekannte Konzepte wiederverwenden
- schon benutzte APIs und Bibliotheken / Dienste berücksichtigen
  - z.B. Benennung und Muster darauf aufbauen

## APIs auf Objektebene

anhand von Beispielen aus dem Buch *API Design* von Kai Spichale

### Grundlagen: Benennung

- Konsistenz, Konsistenz, Konsistenz
  - gleiches Konzept = gleicher Name
  - etablierte Begriffe verwenden
- Klassen: Substantive
  - nach Möglichkeit aus der Domäne, sonst technisch
- Methoden
  - nach Rückgabewert oder Befehl
  - *do*, *get*, *set*, *execute* vermeiden
- Variablen / Parameter
  - keine Abkürzungen
  - keine Typcodierung
- *Ubiquitous Language* (Ubiquitäre Sprache)
  - Sprache zwischen Domänenexperten und Software-Entwicklern

#### Beispiele:

1. Name einer Klasse, die <ul style="list-style-type: none"><li>a. einen Stuhl darstellt?</li><li>b. ein Auto darstellt?</li></ul>	1. Antwort <ul style="list-style-type: none"><li>a. Chair</li><li>b. Car</li></ul>
2. Name einer Methode, die <ul style="list-style-type: none"><li>a. die Musik lauter macht?</li><li>b. die Musik leiser macht?</li><li>c. die einen dreibeinigen Stuhl zurückgibt?</li></ul>	2. Antwort <ul style="list-style-type: none"><li>a. increaseVolume</li><li>b. decreaseVolume</li><li>c. threeLaggedChair</li></ul>

# Typisierung

```
String findArticle( String articleId );  
Article foo( ArticleId bar );
```

Ein Typ erzwingt mehr als 1.000 Worte.

Typisierung kann temporäre Kopplung auflösen:

```
class DocumentCreator {  
    Document create(DocumentDescription desc) { ... }  
    List<Messages> getErrorMessages() { ... }  
}
```

```
class DocumentCreator {  
    DocumentCreatorResult create(DocumentDescription desc);  
}  
  
class DocumentCreatorResult {  
    Document getDocument();  
    List<Messages> getErrorMessages();  
}
```

## NOTE

*Temporäre Kopplung* ist auch bekannt als *semantischer Vertrag*.

**Basistypen sollten keine Kenntnis über Subtypen haben.**

Datenkapselung / Information Hiding beachten

```

class Message {
    private int severityLevel;
    private String msg;

    public Message(int severityLevel,
                   String msg) {
        this.severityLevel =
severityLevel;
        this.msg = msg;
    }

    public String toString() {
        if(severityLevel==1)
            return "INFO: " + msg;
        return "WARN: " + msg;
    }
}

```

```

class Info extends Message {
    public Info(String msg) {
        super(1, msg);
    }
}

class Warning extends Message {
    public Warning(String msg) {
        super(2, msg);
    }
}

```

#### NOTE

Obiges Beispiel ist eine *semantische(!)* Verletzung der Datenkapselung - keine technische. Die Basisklasse hat Wissen implementiert über die Subklassen. Eine gute API vermeidet dies, da - zumindest in Java in diesem Beispiel - weitere Subklassen durch den Nutzer nicht ausgeschlossen werden können. So wird eine falsche Benutzung der API erschwert.

Ein weiteres Beispiel einer semantischen Verletzung der Datenkapselung wäre das Auslassen vom Aufruf einer *init*- Methode, weil man *weiß*, dass diese implizit in einer anderen Methode aufgerufen wird. Ändert sich dann aber die andere Methode, funktioniert der eigene Code nicht mehr (richtig).

## Typisierung: Beispiel SpeedLimit

```

/**
 * Checks the given speed against road traffic regulations.
 * Returns a negative integer, zero or a positive integer as the argument is below
 * speed limit,
 * is within tolerance limit, or exceeds the speed limit.
 */
int checkSpeed(int kmPerHour);

```

**Verbesserung mit typisiertem Rückgabewert:**

```
enum SpeedCheckResult {
    BELOW_SPEED_LIMIT,
    WITHIN_TOLERANCE_LIMIT,
    EXCEEDS_SPEED_LIMIT;
}
```

```
SpeedCheckResult checkSpeed(int kmPerHour);
```

#### NOTE

Über 4.000.000 Rückgabemöglichkeiten reduzieren sich auf 3. Zudem kann der Benutzer jetzt mit dem Rückgabewert etwas anfangen, ohne die Dokumentation lesen zu müssen - in diesem Fall könnte sich die Dokumentation sogar ganz erledigt haben, je nachdem wie der genaue Einsatzzweck der API ist (z.B. manche Zertifizierungen schreiben Dokumentation vor).

### Verbesserung mit typisiertem Parameter:

```
final class Speed {
    private final int kmPerHour;

    public Speed(int kmPerHour) {
        this.kmPerHour = kmPerHour;
    }
    // ...
}
```

```
SpeedCheckResult checkSpeed(Speed vehicleSpeed);
```

#### NOTE

Eine falsche Benutzung der API wird erschwert, da jetzt nicht mehr aus Versehen irgendein anderer Wert als eine Fahrzeuggeschwindigkeit übergeben werden kann (z.B. der Kilometerstand oder ähnliches).

## Typisierung: Beispiel Font

```
textField.setFont("Helvetica", true, true);
```

- (+) Schriftart ist klar
- (-) Schriftart als String und damit anfällig für Schreibfehler
- (-) `true, true` am Ende ist unklar

### Verbesserung mit typisierten Parametern:

```
import com.example.FontType;
import com.example.FontFormat;

textField.setFont(FontType.HELVETICA, FontFormat.BOLD, FontFormat.ITALIC);
```

- (+) Schreibfehler nicht mehr möglich
- (o) 3 einzelne Parameter *seltsam* zum Benutzen

#### Verbesserung mit einem typisierten Parameter:

```
textField.setFont(new FontBuilder()
    .type(FontType.HELVETICA)
    .bold()
    .italic()
    .build());
```

- (+) über den **Builder** leicht zu erweitern (z.B. mit *underlined()*)
- (+) sinnvolle Defaults sind im **Builder** möglich

## Minimale Sichtbarkeit

- API möglichst klein
  - neue Elemente hinzufügen ist einfach
  - vorhandene Elemente entfernen könnte Kompatibilitätsprobleme haben
- nur notwendige Sachen exponieren

#### Richtlinien in Java:

- Klassen: standardmäßig *package-private*
  - *private* ist als *Inner Class* möglich
  - generell *Interfaces* anstatt Klassen exponieren
- Methoden: standardmäßig *private*
  - nur wenige Methoden *protected* oder *public*
- Felder: standardmäßig *private*
  - Ausnahme bei Konstanten (i.d.R. *public* und *static*)

## NOTE

- *private*: nur die Klasse selbst hat Zugriff
- *package-private*: kein *access modifier* → Klassen innerhalb des gleichen Packages haben Zugriff
- *protected*: wie *package-private* aber zusätzlich haben auch abgeleitete Klassen Zugriff
- *public*: jeder hat Zugriff

## Richtlinien in Java (fortgesetzt):

- Packages: generell in Implementierung und API unterteilen
  - Beispiel: Standardbibliothek nutzt die API-Packages `java` und `javax`, die Implementierung findet sich u.a. in `com.sun.unc` `com.oracle`.
- Module: mit Java 9 und Project Jigsaw lässt sich genau festlegen, welche Teile exponiert werden

## Minimale Sichtbarkeit: Beispiel

```
-> com.example.application
    |--> .api
        |--> ObjectInterface.java
        |--> AnotherInterface.java
    |--> .internal
        |--> SimpleObject.java
        |--> SpecializedObject.java
        |--> SimpleOther.java
```

Problem: Die Implementierungen aus `internal` sollen nicht exponiert werden. Wie soll aber der Benutzer diese sonst instanziiieren?

## NOTE

Im obigen Beispiel sind die Interfaces unter `api` *public* und die Implementierungen entweder *package-private* oder *protected*.

## Schnelle 'provisorische' Lösung:

```
-> com.example.application
    |--> .api
        |--> ObjectInterface.java
    |--> .internal
        |--> SimpleObject.java
        |--> SpecializedObject.java
        |--> ObjectBuilder.java
```

## NOTE

`ObjectBuilder` wäre in diesem Beispiel *public*. Es ist keine schöne Lösung, da `ObjectBuilder` im Package *internal* ist, aber immerhin sind die Implementierungen versteckt.

## Erweiterte Lösung:

```
-> com.example.application
    |--> .api
        |--> ObjectInterface.java
        |--> ObjectBuilder.java
    |--> .internal
        |--> SimpleObject.java
        |--> SpecializedObject.java
        |--> ObjectBuilderInternal.java
```

## Implementierungen der Builder:

```
public class ObjectBuilderInternal {

    protected ObjectBuilderInternal() {}

    protected ObjectInterface simpleObjectInternal() {
        return new SimpleObject();
    }

    protected ObjectInterface specializedObjectInternal() {
        return new SpecializedObject();
    }
}
```

```
public class ObjectBuilder extends ObjectBuilderInternal {

    private static ObjectBuilder internalBuilder = new ObjectBuilder();

    public static ObjectInterface simpleObject() {
        return internalBuilder.simpleObjectInternal();
    }

    public static ObjectInterface specializedObject() {
        return internalBuilder.specializedObjectInternal();
    }
}
```

## NOTE

Alternativ kann natürlich auf das *static* verzichtet werden und die *\_Internal\**-Methoden von den exponierten Methoden direkt aufgerufen werden.

*ObjectBuilderInternal* ist zwar immer noch nach außen sichtbar jedoch ohne Funktionalität, außer, der Benutzer leitet von dem Builder ab. Wer jedoch von einem Package, das *internal* im Namen trägt und einer Klasse, die dies ebenfalls tut, ableitet, muss damit rechnen, dass sich diese Klasse ändert und Funktionalität bricht.

## Hilfsmethoden

- können Komfort erhöhen
  - z.B. *addAll* und *removeAll*
- sollten nur allgemein sinnvolle Aufgaben haben
  - z.B. *addAllEven* wäre kein guter Kandidat
- können in *Utility/Helper*-Klassen ausgelagert werden
  - allgemein ist das ein Antipattern

## Optionale Rückgabewerte

- manche Rückgabewerte können oder können nicht vorhanden sein
  - z.B. eine Query in der JPA
  - `Article article = em.find(Article.class, id);`
  - Welche Möglichkeiten hat man, damit umzugehen?

### *Null* zurückgeben:

- Client muss auf *null* vorbereitet sein

```
Article article = em.find(Article.class, id);
if(article==null) {
    // Artikel ist nicht vorhanden
} else {
    // Artikel ist vorhanden
}
```

### Verbesserung: Methode umbenennen in *getByIdOrNull*, um explizit auf *null* hinzuweisen.

## NOTE

Das explizite Hinweisen ist zwar gut, aber generell ist es empfehlenswert auf *null* zu verzichten.

### Exception werfen:



```
Article getById(String id) {
    Article article = em.find(Article.class, id);
    if (article == null) throw new ArticleNotFoundException();
    return article;
}
```

- (+) *fail fast* ist generell gut
- (-) in manchen Fällen kann der Client nicht wissen, ob das Objekt existiert → er muss trotz korrektem Code auf eine Exception vorbereitet sein
- Verbesserung: zusätzlich eine *contains(id)* Methode anbieten
  - (-) semantische Kopplung

### ***NullObject* zurückgeben:**

```
public Discount getById(String id) {
    Discount discount = discounts.getByIdOrNull(id);
    if(discount == null) {
        return new NullDiscount();
    }
    return discount;
}
```

- *NullObject* muss Sinn machen
  - wenn im Client auf *NullObject* anstatt auf *null* geprüft wird, macht es keinen Sinn
- könnte z.B. auch eine leere Liste / Array sein

```
abstract class Discount {
    public abstract Price computeSalePrice(Price originalPrice);
}

class MembershipDiscount {
    public Price computeSalePrice(Price originalPrice) {
        // Berechnung des Verkaufspreises für Mitglieder
    }
}

class NullDiscount {
    public Price computeSalePrice(Price originalPrice) {
        // keine Preisveränderung
        return originalPrice;
    }
}
```

### **Ergebnisobjekt zurückgeben:**

```
public Optional<Discount> getById(String id) {
    Discount discount = discounts.getByIdOrNull(id);
    return Optional.ofNullable(discount);
}
```

- Ergebnisobjekt sollte *mehr* als *NullObjekt* sein
  - z.B. `Discounts.getById(id).ifPresent(value /do stuff/);`

#### NOTE

Das Java `Optional` bringt zwar erweiterte Funktionalität mit (z.B. `isPresent()` oder `orElseThrow()`), lässt sich jedoch nicht gut verketteten.

#### Fazit:

- es gibt nicht **die** Lösung für optionale Rückgabewerte
- es ist immer situationsabhängig, was am besten geeignet ist

## Exceptions

#### Ausnahmearten:

- Exceptions aufgrund Programmierfehler
  - `NullPointerException`, `IndexOutOfBoundsException`, ...
- Exceptions aufgrund falscher Nutzung; *Brechen des API-Vertrags*
  - `EmailNotValidException`, ...
- Exceptions aufgrund Ressourcenausfalls
  - Netzwerkausfall, Speicherplatzmangel, ...

#### NOTE

Je nach Ausnahme, kann der Client versuchen, die Exception zu behandeln und weiter zu machen. Bei den ersten beiden Arten ist dies wahrscheinlicher als bei der letzten (Ressourcenausfall).

#### Checked vs Unchecked Exceptions:

- *Checked Exceptions* müssen abgefangen oder in der Signatur deklariert werden
- *Unchecked Exceptions* können (müssen aber nicht!) abgefangen werden
- generell sollten nur *Unchecked Exceptions* eingesetzt werden
  - u.a. Empfehlung von Joshua Bloch, ein ehemaliger Hauptentwickler der Java-Plattform
  - **niemals** *Checked Exceptions* einsetzen, wenn sich Clients nicht davon erholen können
- Exceptions auf der gleichen Abstraktionsebene wie 'der Rest'
  - Geschäftslogik sollte sich nicht mit *SQLExceptions* befassen müssen
  - ggf. Exceptions konvertieren

- alle Exceptions (*unchecked* und *checked*) sollten dokumentiert sein
  - andere Methoden können sich dann daran orientieren
- generell versuchen Exceptions zu vermeiden
  - z.B. mit zusätzlichen Hilfsmethoden, mit denen vorher geprüft werden kann

Exceptions vermeiden:

```
if( Integer.isInteger(s) ) {
    int num = Integer.parseInt(s);
}
```

- String wird geprüft, ob er ein Integer-Wert ist
- nur falls ja, wird er geparkt (was sonst zu einer Exception führen würde)

## Interfaces

- Interfaces definieren Typen
  - im erweiterten Sinne auch *Eigenschaften*
- ein Objekt kann mehrere Interfaces haben
  - mehrere Rollen sind so möglich
- Interfaces = flache Hierarchien (anders als Vererbung)
- generell: Adjektive als Namen
- lieber Interfaces als Klassen nach außen freigeben
  - man bleibt flexibler, da man nachträglich andere Implementierungen anbieten kann
  - man macht sich mehr Gedanken bei der Deklaration
- nicht-triviale Interfaces, die vom Client implementiert werden sollen, ggf. mit einer abstrakten Basis-Implementierung ausliefern

### Funktionale Interfaces

- Interface mit genau einer abstrakten Methode
  - *Default* Methoden erlaubt
- dienen zur Deklaration von Lambdas

```
public interface Consumer<T> {
    /**
     * Performs this operation on the given argument.
     * @param t the input argument
     */
    void accept(T t);
}
```

Code tendiert dazu, leserlicher zu werden.

## Flag-Interfaces

- leere Interfaces
- dienen zur Markierung bestimmter Eigenschaften
  - das Interface selbst kann diese Eigenschaft nicht erzwingen
- z.B. *Serializable*
- zur Compilezeit können so bestimmte Fehler aufgedeckt werden

## Fluent Interfaces

- interne DSL
  - bessere Lesbarkeit
  - vereinfachte Benutzung
  - bessere fachliche Abstraktion
- Begriff geprägt durch Eric Evans und Martin Fowler

### NOTE

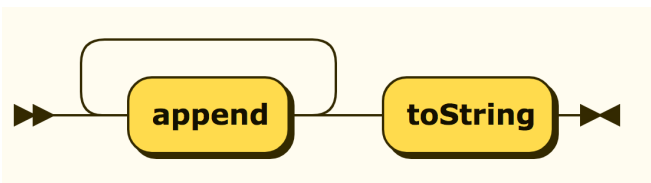
- DSL = Domain Specific Language (domänenspezifische Sprache)
- Eric Evans = Autor des Buchs *Domain Driven Design* und Vorreiter in diesem Bereich
- Martin Fowler = bekannter Software-Architekt und Buchautor

## Beispiel eines Fluent Interfaces

Java Code:

```
new StringBuilder()  
    .append("Hallo")  
    .append(", ")  
    .append("Welt!")  
    .toString();
```

Railroad Diagramm:



EBNF:

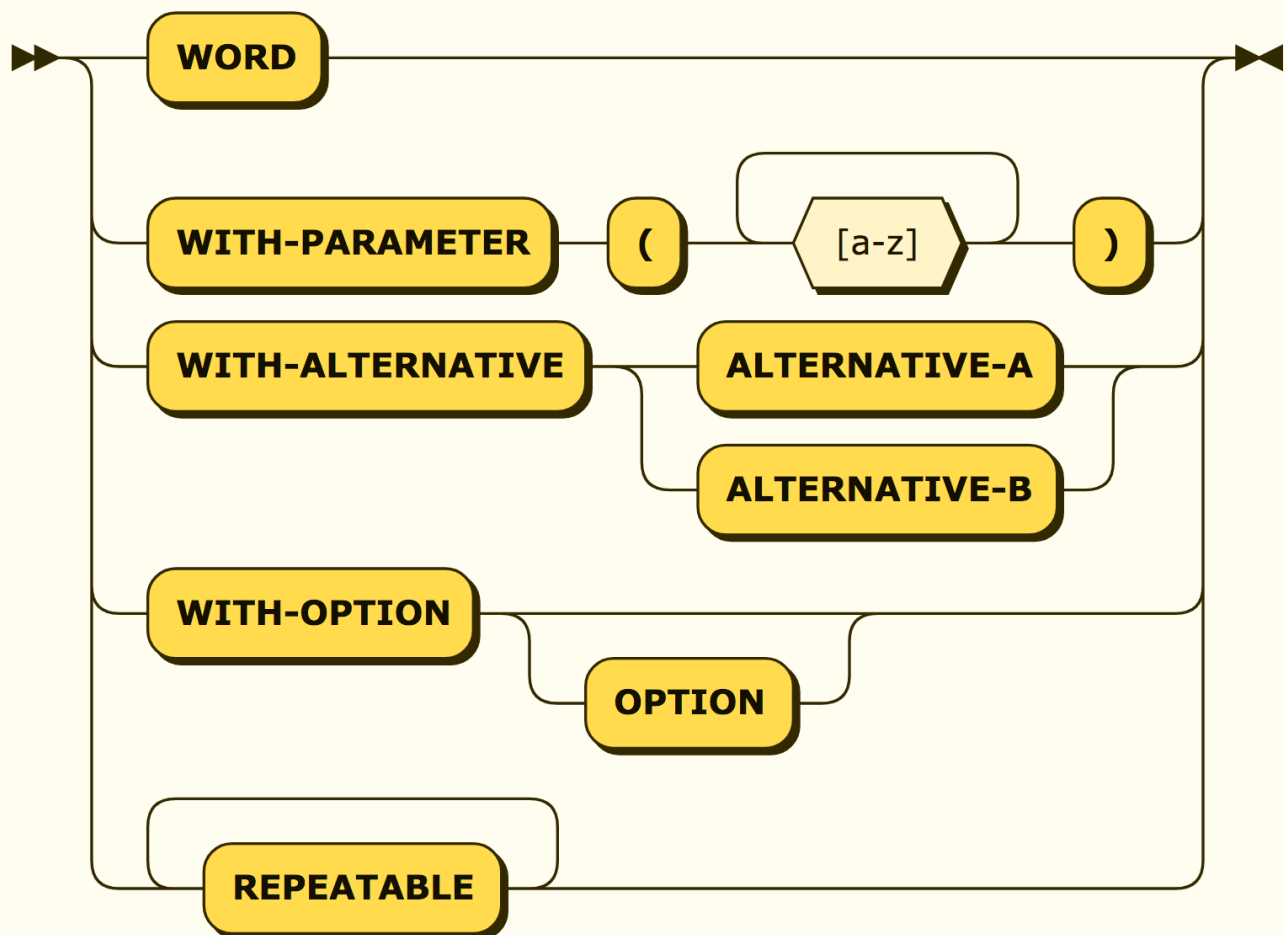
```
Grammar ::= 'append'+ 'toString'
```

### NOTE

- EBNF = erweiterte Backus-Naur-Form
- Diagramme wurden erstellt mit: <http://www.bottlecaps.de/rr/ui>
- die End-Funktion (hier: *toString*) ist optional bei der Darstellung

## Basiskomponente einer DSL

```
Grammar ::= 'WORD'  
        | 'WITH-PARAMETER' '(' [a-z]+ ')'  
        | 'WITH-ALTERNATIVE' ( 'ALTERNATIVE-A' | 'ALTERNATIVE-B' ) | 'WITH-OPTION'  
        'OPTION'?  
        | 'REPEATABLE'+
```



```
interface Begin {  
    End word();  
    End withParameter(String parameter);  
    Alternative withAlternative();  
    Option withOption();  
    NoToMany repeatable();  
}
```

```
interface End {  
    void end();  
}
```

```
interface Alternative {  
    End alternativeA();  
    End alternativeB();  
}
```

```
interface Option extends End {  
    End option();  
}
```

```
interface NoToMany extends End {  
    NoToMany repeatable();  
}
```

## Beispielanwendung der allgemeinen DSL

```

Begin begin = ...

// ein einzelnes Wort und Abschluss
begin.word().end();

// Wort mit Parameter und Abschluss
begin.withParameter("param").end();

// Wort gefolgt von Auswahl und Abschluss
begin.withAlternative().alternativeA().end();
begin.withAlternative().alternativeB().end();

// Wort gefolgt von optionalem Wort und Abschluss
begin.withOption().end();
begin.withOption().option().end();

// wiederholbare Aufrufe und Abschluss
begin.repeatable().repeatable().end();

```

### Beispielanwendung bei Tests:

#### Nicht fließend

```

assertNotNull(text);
assertTrue(text.startsWith("a"));
assertTrue(text.endsWith("z"));
assertTrue(text.isLowerCase());

```

#### Fließend

```

assertThatString(text)
    .NotNull()
    .startsWith("a")
    .endsWith("z")
    .isLowerCase();

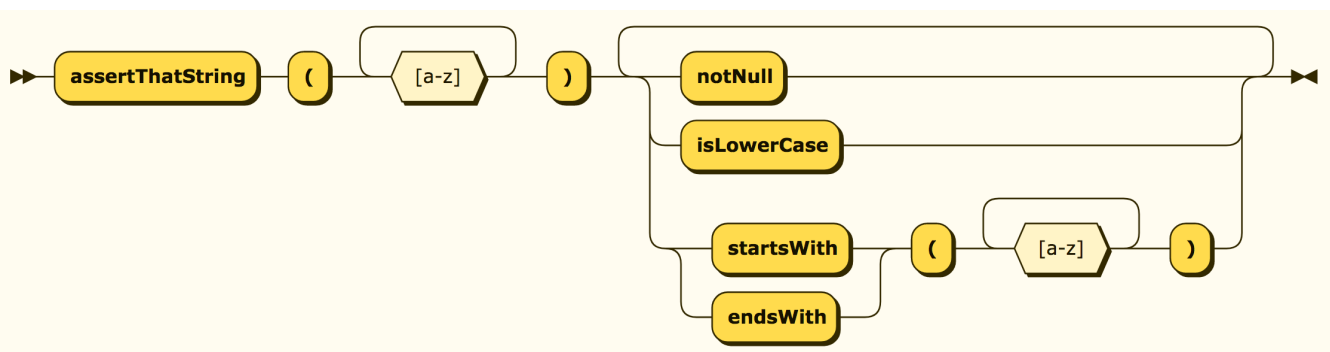
```

### EBNF und Railroad-Diagramm zum Fluent-Test

```

Grammar ::= 'assertThatString' '(' '[a-z]+' ( 'notNull'
    | 'isLowerCase'
    | 'startsWith' '(' '[a-z]+'
    | 'endsWith' '(' '[a-z]+' ) )+

```



## Beispiel JOOQ

```
create.select(CUSTOMER.FIRST_NAME, CUSTOMER.LAST_NAME, count())
    .from(CUSTOMER)
    .join(ORDER).on(CUSTOMER.ID.equal(ORDER.CUSTOMER_ID))
    .where(ORDER.CREATED.ge(date("2016-01-01")))
    .groupBy(CUSTOMER.FIRST_NAME, CUSTOMER.LAST_NAME)
    .having(count().gt(3))
    .orderBy(CUSTOMER.LAST_NAME.asc().nullsFirst())
```

```
SELECT CUSTOMER.FIRST_NAME, CUSTOMER.LAST_NAME, COUNT(*) FROM CUSTOMER
JOIN ORDER ON CUSTOMER.ID = ORDER.CUSTOMER_ID
WHERE ORDER.CREATED >= DATE '2016-01-01'
GROUP BY CUSTOMER.FIRST_NAME, CUSTOMER.LAST_NAME
HAVING COUNT(*) > 3
ORDER BY CUSTOMER.LAST_NAME ASC NULLS FIRST
```

### NOTE

Das Fluent-Interface stellt eine korrekte Syntax sicher - syntaktisch falsches SQL ist nicht möglich → sehr großer Vorteil.

## Alternative zur Methodenverkettung: Schachtelung

```
assertThat(text,
    allOf(
        notNullValue(),
        startsWith("a"),
        endsWith("z"),
        isLowerCase()));
```

- (+) durch eigene Implementierungen kann das Framework erweitert werden (der eigene Typ wird als Parameter übergeben)
- (-) nicht so leicht zu lesen

## Fluent Interfaces: Fazit

- (+) stellen korrekte Syntax sicher
- (+) intuitiv(er) zu nutzen
- (-) deutlich erhöhter Programmieraufwand
  - Kosten / Nutzen muss im Verhältnis stehen

## Template-Methoden

- Erweiterungspunkte für das Framework

- werden vom Benutzer implementiert
- API (im engeren Sinn): Zugriff auf implementierte Funktionalität
  - Erweiterung der API ist (normal) kompatibel für den Benutzer
- SPI: Möglichkeit das Verhalten zu ändern
  - Erweiterung eines SPIs ist (normal) nicht kompatibel

**NOTE**      SPI = Service Provider Interface

### Beispiel für eine Template-Methode

```
abstract class OrderSorter {
    private List<Order> orders;

    public void sort() {
        ...
        if(compare(o1,o2)) {
            ...
        }

        /**
         * Compares its two arguments for order. Returns a negative
         * integer, zero, or a positive integer as the first argument
         * is less than, equal to, or greater than the second.
         */
        abstract int compare(Order o1, Order o2);
    }
}
```

(-) API-Funktionalität und API-Erweiterung (SPI) werden vermischt

### Funktionalität und Erweiterung getrennt

```
public class OrderComparator implements Comparator<Order> {
    public int compare(Order o1, Order o2) {
        return o1.getCreationDate().compareTo(o2.getCreationDate());
    }
}
```



```
public final class OrderSorter {
    private final Comparator<? super Order> comp;
    public OrderSorter(Comparator<? super Order> comp) {
        this.comp = comp;
    }

    public void sort(List<Order> orders) {
        Collections.sort(orders, comp);
    }
}
```

### Vorteile der Trennung

- Erweiterung (*OrderComparator*) ist spezialisierter und kann an anderen Stellen verwendet werden
- ursprüngliche Klasse (*OrderSorter*) ist nicht mehr *abstract* und kann *final* sein
  - nicht erweiterbar bedeutet mehr Kontrolle für den API-Designer
  - Erweiterungen / Änderungen sind einfach möglich → keine / überschaubare Nebeneffekte für den Benutzer

### Parametererweiterung bei Template-Methoden

- die API ändert sich und die Parameter müssen geändert oder erweitert werden
- nach Möglichkeit sollte diese Änderung kompatibel sein

```
public interface ServiceProviderInterface {
    void callbackMethod(String param);
}
```

(-) Änderung der Parameter bedeutet im obigen Beispiel immer Inkompatibilität für den Client

#### NOTE

Der Client sollte möglichst nie gezwungen werden, vorhanden Code anzupassen oder mit für ihn unnötigen Funktionen erweitern (was hier passieren würde, wenn man die Methode überladen würde).

### Lösung: Parameter-Objekt

```
public class ParameterObject {
    public String foo() { ... }
    public Integer bar() { ... }
}

public interface ServiceProviderInterface {
    void someMethod(ParameterObject param);
}
```

Das Parameter-Objekt (auch Context-Objekt) kann kompatibel erweitert werden.

## Callbacks

- ein Objekt / eine Funktion wird registriert und (bei Bedarf) aufgerufen
- synchron oder asynchron
- ***Inversion of Control***
  - Framework entscheidet, wann Client aufgerufen wird

### Synchrone Callbacks

Beispiel:

```
List<String> myList = new ArrayList<>();  
myList.add('hello');  
myList.add('world');  
  
myList.forEach(listItem ->  
    System.out.println(listItem)  
);
```

- Aufrufer ist blockiert, bis der Callback vollständig durchgeführt wird
  - hier: bis der Callback für jedes Element in der Liste durchgeführt wurde
- geeignet, wenn man das Ergebnis direkt braucht

### Asynchrone Callbacks

- realisiert mit *Pull*, *Poll* oder *Push*

Beispiel *Pull*:

```
AsynchronousSocketChannel ch = ...;  
ByteBuffer buf = ...;  
Future<Integer> handle = ch.read(buf);  
Integer result = handle.get();  
// Integer result = handle.get(5, TimeUnit.SECONDS);
```

*Pull* blockiert zwar irgendwann, jedoch entscheidet der Client darüber.

Beispiel *Poll*:

```
// in eigenem Thread
while (!handle.isDone()) {
    Thread.sleep(1000)
}
// benachrichtige jemand oder fahre Operation aus
```

- (+) keine Blockade
- (-) fehleranfällig
- (-) viel Boiler-Plate-Code

Beispiel *Push*:

```
class Handler implements CompletionHandler<Integer,Connection> {
    public void completed(Integer result, Connection conn) {
        // Ergebnis behandeln
    }

    public void failed(Throwable exc, Connection conn) {
        // Fehlerbehandlung
    }
}
```

```
ch.read(buffer, connection, handler);
```

- (+) keine Blockade
- (+) einfach zum Implementieren
- (+) einfach zum Lesen

#### NOTE

Mit Java 8 ist die Klasse *CompletableFuture* dazu gekommen, die quasi die Verschmelzung von *Future* mit *CompletionHandler* ist. Das ist die aktuell bevorzugte Variante für neue APIs / Programme.

### Fazit Callbacks

- ***Inversion of Control***
- synchron, wenn das Ergebnis gleich gebraucht wird
- asynchron bei allen anderen Operationen
  - nach Möglichkeit die Push-Variante einsetzen

## Annotationen

- Metainformationen für Code-Blöcke

- z.B. können bestimmte Direktiven beim Kompilieren ausgeführt werden
- z.B. kann während der Laufzeit etwas mit annotierten Objekten gemacht werden

Beispiel von RESTeasy:

```
@Path("/library")
public class Library {

    @GET
    @Path("/books")
    public String getBooks() {...}

    @GET
    @Path("/book/{isbn}")
    public String getBook(@PathParam("isbn") String id) {
        // search my database and get a string representation and return it
    }
}
```

#### NOTE

Annotationen sind zwar mächtig, aber nicht trivial in der Verarbeitung. In [Project Lombok](#) empfiehlt es sich, reinzuschauen, wenn Interesse hierfür da ist.

## Immutability

- unveränderliche Objekte (nach der Erzeugung)
- Ableitung der Klasse nicht möglich (*final*)
- Felder sind *final* und entweder *private* oder selbst *immutable*

Vorteile:

- Thread-sicher
- Wiederverwendbarkeit
- einfach(er) zu benutzen

Nachteil: für jeden unterschiedlichen Wert, gibt es ein (neues) Objekt.

#### NOTE

Sie sind einfacher zu benutzen, weil sie nur einen Zustand haben - den bei der Erzeugung.

Veränderbare Objekte müssen ggf. mehrfach auf ihren Zustand geprüft werden.

### Beispiel Immutability und Thread-Sicherheit

```

class Point {
    private int x, y;
    void setCoordinates(int x, int y) {
        synchronized (this) {
            this.x = x;
            this.y = y;
        }
    }

    synchronized int getX() { return x; }
    synchronized int getY() { return y; }
}

```

```

Point p = ...;
synchronized (p) {
    int x = p.getX();
    int y = p.getY();
}

```

Mit unveränderlichen Objekten ist dieses Problem implizit gelöst.

### Beispiel Immutability und Wiederverwendbarkeit

```

String s1 = "test";
String s2 = new String(s1);
String s3 = "test";
System.out.println(s1 == s2); // false
System.out.println(s1 == s3); // true

```

- *String* ist eine sehr häufig verwendete Klasse
- durch die Unveränderlichkeit können Instanzen einfach wiederverwendet werden

## Threadsicherheit

- parallele Abarbeitung ohne Seiteneffekte
- ist nur Thema bei *statusbehafteten* und/oder *veränderbaren* Objekten

Vorteile:

- Ausnutzen von mehreren Kernen
- schnelleres Programm
- nicht-blockierende Prozesse

**Nachteil: alles andere als trivial umzusetzen.**

```
public class Counter {
    private int value;

    public void increment() {
        value++;
    }
}
```

Ist dieser *Counter* threadsicher?

Nein, da *value* erst gelesen und dann inkrementiert wird.

#### NOTE

Rufen zwei Threads gleichzeitig *increment* auf, könnte es passieren, dass sie beide den gleichen Wert lesen, bevor sie inkrementieren (und *value* damit effektiv nur um 1 inkrementiert wird anstatt um 2).

### Überarbeiteter *Counter*

```
public class SafeCounter {
    private int value;

    public synchronized void increment()
    {
        value++;
    }
}
```

```
public class SafeCounter {
    private final AtomicInteger value
        = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet();
    }
}
```

```
public class StatelessCalculator {
    public int add(int a, int b) {
        int result = a + b;
        return result;
    }
}
```

Ist dieser *Calculator* threadsicher?

Ja, da statuslos.

# Kompatibilität

von Java-APIs

# Code-Kompatibilität

- die einfachste Art der Kompatibilität
- was mit Version 1.0 kompiliert, kompiliert mit Version 1.1
- generell inkompatibel: entfernen von (öffentlichen) Methoden und Klassen
- Achtung bei *offenen* Klassen
  - wenn der Benutzer abgeleitet hat und die neue Version eine gleichnamige Methode erweitert

## Binäre Kompatibilität

- binäre Version 1.0 kann von der binären Version 1.1 ersetzt werden
- Achtung bei
  - primitiven Konstanten: diese werden zur *Compilezeit* ersetzt
  - überladenen statischen Methoden: auch hier wird zur Compilezeit aufgelöst

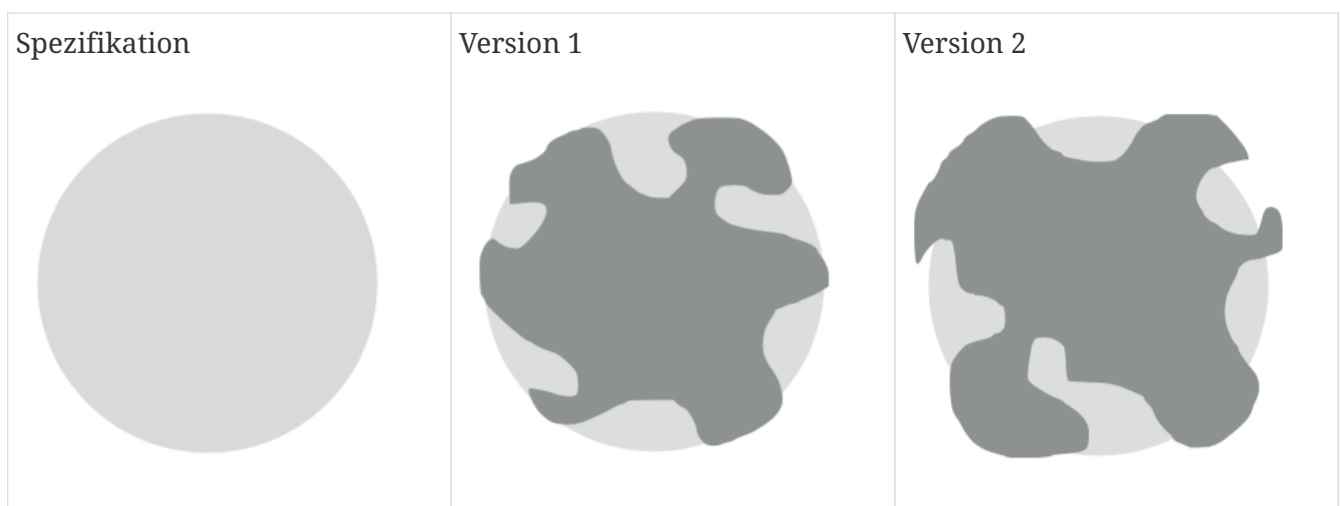
### NOTE

Das Auflösen zur Compilezeit bedeutet, dass z.B. die Konstante `MY_CONST = 1` überall im Code mit `1` ersetzt wird. Wenn der Client eine neue API-Version als binäre Datei einbindet - und diese neue Version `MY_CONST = 2` definiert - dann hat der Client in seinem Code trotzdem noch die `1` stehen, da nicht neu kompiliert wurde.

## Funktionale Kompatibilität

Erwartetes Verhalten bleibt in neuen Versionen gleich.

Amöben-Effekt:



Bilder-Quelle: *API Design* von Kai Spichale

Amöben-Effekt kann folgende Ursachen haben:

NOTE

- nicht vollständig umgesetzte Spezifikation
- nicht-spezifizierte Funktionalität
- Fehlerverbesserung
- Refactorings

## Verwandschaftsbeziehung

- Kompatibilität steht immer im Verhältnis zu einer anderen API-Version
- Abwärtskompatibel: neuere Version kann ältere Version ersetzen
- Vorwärtskompatibel: ältere Version kann neuere Version ersetzen

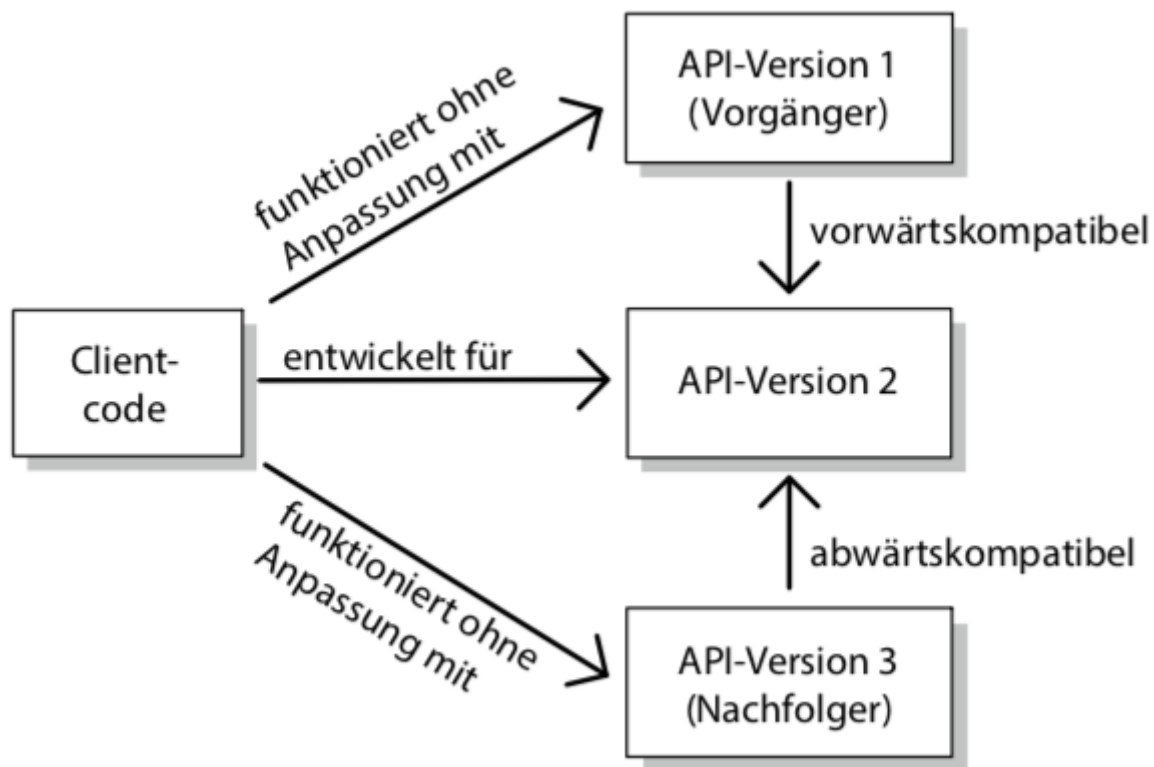


Bild-Quelle: *API Design* von Kai Spichale

## Design By Contract

- Vertrag regelt das Zusammenspiel
- *Invarianten*: Bedingungen für alle Objekte einer Klasse zu jeder Zeit (außer während eines Methodenaufrufs)
- *Vorbedingung*: muss vom **Aufrufer** eingehalten werden
- *Nachbedingung*: muss vom **Aufgerufenem** eingehalten werden



Bedingung	Änderung	Auswirkungen für Aufrufer	Auswirkungen für Implementierer
Vorbedingungen für Methoden	Verstärkung	inkompatibel	kompatibel
	Abschwächung	kompatibel	inkompatibel
Nachbedingungen für Methoden	Verstärkung	kompatibel	inkompatibel
	Abschwächung	inkompatibel	kompatibel
Invarianten für Felder	Verstärkung	kompatibel für Getter-Rolle	inkompatibel für Setter-Rolle
	Abschwächung	inkompatibel für Getter-Rolle	kompatibel für Setter-Rolle

Bild-Quelle: *API Design* von Kai Spichale

### Beispiel: Änderung der Vorbedingung

```
/** @param collection a non-null collection */
public boolean bookAll(Collection collection);
```

```
public boolean bookAll(Collection coll) {
    int size = coll.size();
    ...
}
```

Verschärfung: *Collection* muss mindestens 1 Element enthalten

- Implementierung hat kein Problem
- Client kann inkompatibel werden

Abschwächung: *Collection* darf auch *null* sein

- für Client kein Problem
- Implementierung erwartete kein *null* und prüft das bisher nicht → inkompatibel

### Beispiel: Änderung der Nachbedingung

```
/** @return returns never null */
public List<Booking> getAll();
```

```
List<Booking> allBookings = getAll();
allBookings.forEach(item -> item.print());
```

Abschwächung: <i>return</i> darf <i>null</i> sein <ul style="list-style-type: none"> <li>• für Implementierung kein Problem</li> <li>• Client ist darauf nicht vorbereitet → inkompatibel</li> </ul>	Verschärfung: <i>return-Collection</i> muss mindestens 1 Objekt enthalten <ul style="list-style-type: none"> <li>• für Client kein Problem</li> <li>• Implementierung muss geändert werden</li> </ul>
--	---

### Beispiel: Änderung von Feldinvarianten

Felder werden immer lesend (get) und schreibend (set) getrennt betrachtet.

```
/** always positive or zero */
private int balance;
```

Abschwächung: negative <i>balance</i> erlaubt <ul style="list-style-type: none"> <li>• für <i>set</i> kein Problem</li> <li>• Aufrufer kann bei <i>get</i> Probleme bekommen, da er nicht vorbereitet ist</li> </ul>	Verschärfung: <i>balance</i> darf höchstens den Wert 1000 haben <ul style="list-style-type: none"> <li>• für <i>get</i> kein Problem</li> <li>• Was macht <i>set</i>, wenn Wert &gt; 1000 übergeben wird? → inkompatibel</li> </ul>
--	---

#### NOTE

Achtung: Die drei obigen Beispiele beziehen sich auf *Design by **Contract***. Der Vertrag wird (erstmal) **nicht** per Code erzwungen.

## Änderungen in der Praxis

- API-Änderungen mind. 1 Version früher ankündigen
  - **@Deprecated** → Benutzer bekommt Warnung bei Benutzung von veralteten API-Teilen
- Delegation
  - alte API-Teile rufen die neuen Teile auf

```
/**
 * @deprecated As of release 1.3, replaced
 * by {@link #getPreferredSize()}
 */
@Deprecated
public Dimension preferredSize() {
    return getPreferredSize();
}
```

#### NOTE

Die alte Methode der Delegation kann natürlich in einer späteren Version entfernt werden - oder sie bleibt parallel erhalten (durch die Delegation ändert sich an der Wartung ändert erstmal nichts und daher ist die parallele Anbietung kein Problem).

## Namenskonventionen bei Interfaces

- Interfaceänderungen führen immer zu Inkompatibilität (außer bei Default-Methoden)
- anstatt das Interface zu erweitern gibt es eine neue Version, die die alte Version erweitert

```
public interface MyInterface {  
    void myMethod();  
}
```

```
public interface MyInterface2  
    extends MyInterface {  
    void newMethod();  
}
```

Die neue API-Version bietet nur noch Implementierung von *MyInterface2* an, ist aber durch die Erweiterung kompatibel zu *MyInterface*. Vorhandene Clients müssen höchstens einen *Downcast* machen.

### NOTE

Das Eclipse-Projekt benutzt oben vorgestellte Namenskonvention. Erweiterte Interfaces bekommen eine fortlaufende Nummer.

## Defaultmethoden

- Interfaces können abwärtskompatibel erweitert werden
- Defaultmethoden können überschrieben werden

```
public interface Iterator<E> {  
    //...  
    default void remove() {  
        throw new UnsupportedOperationException("remove");  
    }  
}
```

### NOTE

Das Interface ist trotz der Exception abwärtskompatibel. Die alten Clients kennen diese neue Methode nicht und haben sie deshalb bisher nicht aufgerufen.

## Extension Interface

- ähnlich wie *Namenskonventionen bei Interfaces*
- erweiterte Interfaces ersetzen **nicht** den alten Typ

```

/* Root interface */
public interface Document {
    Document getExtension(InterfaceId id
);
}

/* Extension interface */
public interface Translatable extends
Document {
    Translateable translate(Language
language);
}

/* Extension interface */
public interface Printable extends
Document {
    void print();
}

```

```

public class SimpleDocument implements
Document, Printable,

Translatable {
    @Override
    public Translatable translate
(Language language) { ... }

    @Override
    public void print() { ... }

    @Override
    public Document getExtension
(InterfaceId id) {
        switch (id) {
            case PRINT: return this;
            case TRANSLATE: return this;
        }
        throw new
IllegalArgumentException(
            "Could not handle id: " + id);
    }
}

```

## Kompatibilitätstest für firmeninterne APIs

```

abstract class Api {
    public static final int VERSION = 1;
    protected Api() {
        init(Api.VERSION);
    }

    protected abstract void init(int version)
        throws new IllegalStateException();
}

```

```

class ClientImpl extends Api {
    protected void init(int version) {
        if(version != Api.VERSION) {
            throw new IllegalStateException();
        }
    }
}

```

Was passiert?

- Konstante *VERSION* wird zur Compilezeit überall ersetzt
- ändert sich die API-Version ohne Client-Neukompilierung: Laufzeitfehler

#### NOTE

Natürlich kann der Test auch für firmen-externe APIs eingesetzt werden, jedoch setzt dies Wissen für den Benutzer voraus.

### Automatisierte Kompatibilitätsprüfung

- *Test Compatibility Kits* prüfen automatisiert
  - z.B. Java Compatibility Kit
- sinnvoll, wenn Kompatibilität gewährleistet sein muss

## REST

### RESTful HTTP

### REST vs HTTP

- REST = **R**epresentational **S**tate **T**ransfer
  - kein Standard, keine Technologie
  - wird v.a. mit HTTP umgesetzt
- HTTP != REST && REST != HTTP
- für die Veranstaltung: REST = RESTful HTTP

## REST-Grundprinzipien

- Ressourcen und deren Repräsentationen als zentrale Bausteine
- Verwendung von Hypermedia
- Verwendung von HTTP-Standardmethoden
- statuslose Kommunikation

## REST-Ressourcen

- Ressourcen haben Repräsentationen
  - z.B. **JSON**, **XML**, HTML, Videos, Bilder, Texte, ...
- unterschiedliche Repräsentationen möglich
  - z.B. JSON und XML
- eindeutig identifizierbar
  - z.B. per URI: <http://api.example.com/users/123>
- Ressourcen-Modell != Domänenmodell

- kann sich aber überschneiden

### Beispiel: Ressource - Repräsentation - URI

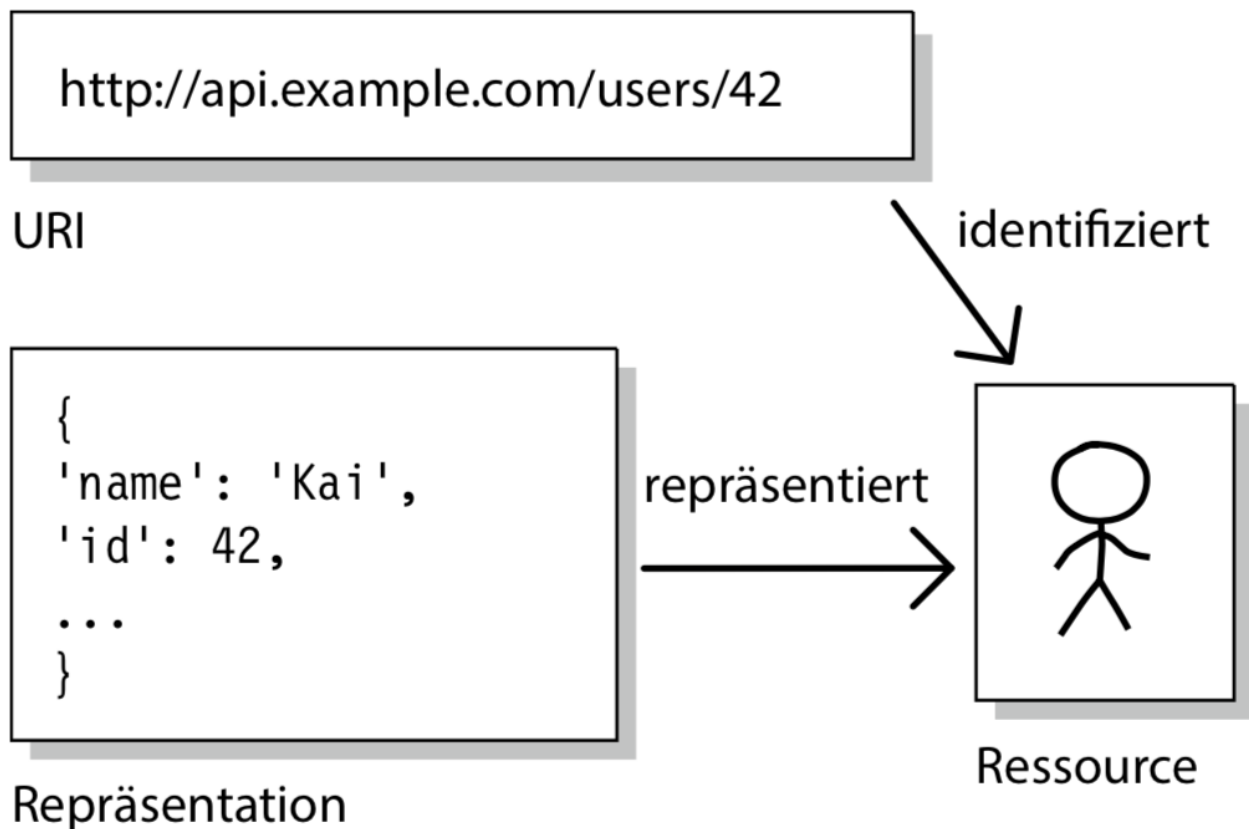


Bild-Quelle: API Design von Kai Spichale, S. 149

## Exkurs: URI vs URL

- URI (Uniform **R**essource **I**dentifier) > URL (Uniform **R**essource **L**ocator)
- URI dient zur Identifikation und URL zum Aufruf
  - bei Web-APIs häufig das Gleiche
- URN (Uniform **R**essource **N**ame) ist ebenfalls eine URI, kann jedoch nicht aufgerufen werden
  - z.B. ISBN eines Buchs

## Ressourcenkategorien

- Einzelressource
  - z.B. <http://api.example.com/products/123>
- Collection-Ressource
  - z.B. <http://api.example.com/products>
- Primärressource
  - z.B. <http://api.example.com/users/678>

- Subresource
  - z.B. <http://api.example.com/users/678/addresses>
  - beliebig tief schachtelbar

## Ressourcen-Benennung

- generell Substantive
- generell Plural
- Einzelressource als Subresource einer Collection-Ressource über die ID
  - z.B. <http://api.example.com/users/678>

### NOTE

Dies sind Daumenregeln und müssen natürlich den gegebenen Bedingungen angepasst werden. Jedoch empfiehlt es sich, sich so weit wie möglich an diese zu halten.

## Hypermedia

- Inhalt, der über sich selbst hinausgeht
  - z.B. HTML mit Hilfe von Links
- Verknüpfung von anderen Ressourcen
- Vorteil: werden nur bei Bedarf geladen

### Beispiel von Hypermedia in XML:

```
<message self="http://example.com/messages/17">
  <body>...</body>
  <attachment ref="http://example.com/attachments/1701" />
</message>
```

Quelle: API Design von Kai Spichale, S. 147

## HTTP-Standardmethoden

- GET
- HEAD
- PUT
- POST
- DELETE
- PATCH
- OPTIONS

- TRACE & CONNECT
  - für die Veranstaltung nicht relevant

## HTTP-Methode: GET

- Leseoperation
- idempotent
  - keine Zustandsänderung
  - keine Seiteneffekte
- Conditional GET
  - **If-Modified-Since**-Header
  - **If-None-Match**-Header
- Partial GET
  - per **Range**-Header nur einen Teil der Ressource abfragen

### NOTE

**If-Modified-Since:** wenn sich die Ressource ab einem bestimmten Datum nicht geändert hat

**If-None-Match:** wenn die Ressource keinen Zustand aufweist, der in den übergebenen ETags hinterlegt ist

### Beispiel: Partial Get

```
GET /pictures/123 HTTP/1.1
Host: api.example.com
Range: bytes=0-9999
```

```
HTTP/1.1 206
Partial Content
Accept-Ranges: bytes
Content-Length: 70000
Content-Range: bytes 0-9999/70000
Content-Type: image/jpeg
{Binary}
```

## HTTP-Methode: HEAD

- Leseoperation
- idempotent
- wie GET nur ohne Body
- z.B. um Meta-Daten einer Ressource abzufragen



- (Video-)Größe, Content-Type, ...

```
HEAD /videos/cat.mp4 HTTP/1.1
Host: funny-videos.example.com
```

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 90000
Content-Type: video/mp4
```

## HTTP-Methode: PUT

- neue Ressource anlegen oder existierende verändern
  - URI der neuen Ressource ist die Aufruf-URI
- idempotent
- Server darf die übermittelten Daten ignorieren, ändern oder ergänzen

## HTTP-Methode: POST

- Hauptzweck: neue Ressource anlegen
- **nicht** idempotent
- URI der neuen Ressource wird vom Server bestimmt
- häufiger *Missbrauch*: Operationen auf dem Server anstoßen

## HTTP-Methode: DELETE

- löschen einer Ressource
- nicht vorhandene Ressource löschen ist erlaubt (kein Fehler)
- **200 »OK«** oder **204 »Accepted«** als Antwortcode

## HTTP-Methode: Options

- mögliche Kommunikationsoptionen
  - z.B. **HEAD**, **GET**, ...
  - nach Möglichkeit im **Allow**-Header

```
HTTP/1.1 200 OK
ALLOW: HEAD, GET, PUT, OPTIONS
```

# HTTP-Methode: Patch

- partielles Aktualisieren einer Ressource
  - **PUT** ersetzt die gesamte Ressource
- nur notwendige Daten werden übertragen
- häufig wird **PUT** dafür eingesetzt

## Objektorientierte Schnittstelle zu REST

Objektorientierte Schnittstelle	RESTful HTTP
getUsers()	GET /users
updateUser()	PUT /users/{id}
addUser()	POST /users
deleteUser()	DELETE /users/{id}
getUserRoles()	GET /users/{id}/roles

Bild-Quelle: API-Design von Kai Spichale

# HATEOAS

## Hypermedia As The Engine Of Application State

## Dynamischer Workflow

- ausgehend von einem Einstiegspunkt folgt man angebotenen Links
- Ressourcen geben weiterführende Links bekannt
  - diese Links können auch dynamisch generiert werden
- keine feste Codierung von URIs im Client notwendig
- hohe *Affordance*

### NOTE

*Affordance*: Maß, wie gut sich ein Objekt selbst beschreiben kann (führt zu intuitiver Nutzung).

## Link-Header

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://api.example.com/>
      rel="self"; type="text/html;charset=UTF-8";
      title="Homepage";
      verb="GET, HEAD, OPTIONS",
<http://api.example.com/videos> rel="all";
      type="application/json;charset=UTF-8";
      title="List of all videos";
      verb="GET"
```

Mit dem **Link**-Header kann HATEOAS zwar umgesetzt werden, beschränkt sich dann aber auf die grundlegenden Funktionen und bringt Einschränkungen mit. Besser sind Hypermedia-Formate wie z.B. **HAL** oder **Siren**.

#### NOTE

- **all**: Link zu einer Collection-Ressource
- **new**: Link zum Anlegen einer Ressource
- **next**: nächster Schritt im Workflow
- **previous**: vorheriger Schritt im Workflow
- **self**: Link auf die aktuelle Ressourcenrepräsentation

Linkrelation	Bedeutung
all	Listenrepräsentation einer Ressource (Collection-Ressource)
new	Anlegen einer neuen Ressource
next	nächster Schritt im Workflow
previous	vorheriger Schritt im Workflow
self	aktuelle Ressourcenrepräsentation

## Fallbeispiel: Webshop

entnommen aus dem Buch *API-Design* von Kai Spichale

### Objekte des Webshops

Konzept	Beschreibung
product (name, id)	Zentrale Objekte des Webshops
picture (name, id)	Produkte können durch Bilder dargestellt werden

Konzept	Beschreibung
review (id, productId, rating, author, comment)	Jedes Produkt kann durch Käufer bewertet werden.
tag (name)	Produkte können mit Tags kategorisiert werden.

## Iteration 1

```
http://api.example.com/getProducts
http://api.example.com/createProduct
http://api.example.com/createReview
```

```
http://api.example.com/updateProduct
http://api.example.com/deleteProduct
http://api.example.com/lockProduct
http://api.example.com/unlockProduct
http://api.example.com/getReviews
http://api.example.com/updateReview
http://api.example.com/removeReview
```

### Vor-/Nachteile:

- (+) sprechende Namen
- (-) URIs entsprechen nicht dem HTTP-Standard
  - widerspricht dem Qualitätsziel: intuitiv zu nutzen
- (-) viele URIs
- (o) Server muss machen, was Client vorgibt

**Iteration 1 entspricht RPC über HTTP.**

## Iteration 2

```
http://api.example.com/products/create?name=Monitor
http://api.example.com/products?method=create&name=Monitor
```

### Vor-/Nachteile:

- (+) sprechende Namen
- (+) wenige URIs
- (-) URIs entsprechen nicht dem HTTP-Standard
  - widerspricht dem Qualitätsziel: intuitiv zu nutzen

- (o) Server muss machen, was Client vorgibt

**Iteration 2 täuscht durch seine Flexibilität über die RPC hinweg.**

**NOTE**

RPC über HTTP ist meist ein Anfängerfehler und resultiert in einer unintuitiven und komplexen API, die sich an keine Web-Standards hält. Zudem wird der Server stark mit dem Client gekoppelt.

Generell sollten man so einen Stil vermeiden.

## Zuordnung: Aktion → HTTP-Methode

Fachliche Aktion	HTTP-Methode
Produkt anlegen	POST (alternativ PUT)
Produkt löschen	DELETE
Produkt aktualisieren	PUT (alternativ POST oder PATCH)
Produkt sperren/freigeben	POST

## Ressourcen identifizieren

Ressourcen werden mit dem Domänenexperten / Kunden identifiziert

- product list
  - a product
    - name
    - sku (stock keeping unit)
    - review list
      - a review
    - picture list
      - a picture

**NOTE**

(sku - stock keeping unit) = Artikelnummer

## Iteration 3

```
http://api.example.com/products
http://api.example.com/products/{SKU}
http://api.example.com/products/{SKU}/reviews
http://api.example.com/products/{SKU}/reviews/{REVIEW_ID}
http://api.example.com/products/{SKU}/pictures
http://api.example.com/products/{SKU}/pictures/{PICTURE_ID}
```

## Test Iteration 3: Produkt(e) abfragen

- gesamte Produktliste (10 Produkte / Seite; ohne Reviews, nur 1 Bild)
  - <http://api.example.com/products> → firstTen → forEach
    - <http://api.example.com/products/{SKU}>
    - <http://api.example.com/products/{SKU}/pictures> → first
      - [http://api.example.com/products/{SKU}/pictures/{PICTURE\\_ID}](http://api.example.com/products/{SKU}/pictures/{PICTURE_ID})
- für jedes Produkt müssen 3 Aufrufe gemacht werden - nicht schön, aber ok

### NOTE

Ein erster Verbesserungsvorschlag wäre, gleich eine Bild-ID mitzuschicken. Dies reduziert die Aufrufe pro Produkt auf 2 Aufrufe anstatt 3.

## Test Iteration 3: Produkt hinzufügen

- **POST** auf <http://api.example.com/products>
  - der Body enthält die Daten
    - Namen
    - ggf. Bild-ID
  - als Response SKU
- woher kommt Bild-ID?
  - normalerweise vom Server generiert

### Lösung:

- erst das Produkt erstellen, dann das Bild hochladen
  - **POST** auf <http://api.example.com/products>
  - **POST** auf <http://api.example.com/products/{SKU}/pictures>
- Problem: neue Anforderung: jedes Produkt **muss** ein Bild haben
  - Inkonsistenz muss vermieden werden
  - z.B. wie verhält sich die API / das System, wenn nur der erste **POST** durchgeführt wird?

### Lösung:

- erst das Bild hochladen, dann Produkt mit Bild-ID erstellen

- Problem: **POST** auf <http://api.example.com/products/{SKU}/pictures> ohne SKU nicht möglich
- Lösung: Ressource ändern
  - <http://api.example.com/pictures> anstatt [/products/{SKU}/pictures](http://api.example.com/products/{SKU}/pictures)
- Anforderungen sind wichtig - spricht mit den Domänenexperten

#### NOTE

Auch bei diesem trivialen Beispiel sieht man, wie wichtig es ist, die genauen Anforderungen zu kennen und die API im **Vorfeld** zu evaluieren.

Wenn z.B. auch die Datenbank so aufgebaut war, dass jedem Bild ein Produkt zugeordnet war (und das Feld als **NOT NULL** deklariert war), müsste man nicht nur die Ressource, sondern auch das Datenbankmodell ändern.

So oder so: ein Kunde / der Chef ist nie begeistert, wenn man erstmal sagt: 'Das geht nicht.' / 'Das kostet extra.'

## Fortgeschritten: Produkt + Bild mit Transaktionen anlegen

- neue Ressource: <http://api.example.com/productconfigurations>
  - **POST** erzeugt neue Ressource
  - neue Ressource kann in beliebiger Reihenfolge befüllt werden (z.B. mit **PUT**)
  - abschließend wird per **POST** auf eine Subressource die Transaktion beendet
    - z.B. <http://api.example.com/productconfigurations/{ID}/release>

## Asynchrone Bearbeitung: Client Polling

- u.U. ist eine asynchrone API erforderlich
  - z.B. dauert das Erzeugen einer neuen Ressource extrem lange
- anstatt **201 - Created** gibt man **202 - Accepted** mit Status-Link zurück
  - Status-Link-Beispiel: <http://api.example.com/receipts/{ID}>
  - Status-Link z.B. im **Location**-Header mitgeben
- Client prüft, ob die Ressource erstellt wurde
  - noch nicht:

```
HTTP/1.1 304 Not Modified
Location: http://api.example.com/receipts/3750527582
```

- erstellt:

HTTP/1.1 303 See Other  
Location: <http://api.example.com/products/0826552>

# URI Design

## Hackable / Hackbar

- jede Ebene sollte eine gültige Ressource sein

```
http://example.shop/products/phones/samsung  
http://example.shop/products/phones  
http://example.shop/products
```

- der Benutzer könnte nur per URIs navigieren
  - gleicher ID-Aufbau

```
https://jira.spring.io/browse/DATAREST-516  
https://jira.spring.io/browse/DATAREST-515  
https://jira.spring.io/browse/DATAREST-514
```

- kennt man eine andere Ticket-Nummer, kann man einfach die URI abändern

### Vor-/Nachteile:

- (+) intuitiv zu nutzen
- (+) Fehler leichter zu sehen
- (-) u.U. Sicherheitsrisiko

## Kurz

- so kurz wie möglich, so lang wie nötig
- trotzdem verständlich
- <http://api.example.com/products/23/1> ist zwar kurz, aber nicht verständlich
  - besser: <http://api.example.com/products/23/reviews/1>

## Groß-/Kleinschreibung

- Domain-/Servername unabhängig von Groß-/Kleinschreibung
  - restliche URI nicht
- generelle Empfehlung: alles klein



- Großschreibung führt zur gleichen Ressource

## Sonderzeichen

- a-z, 0-9 und Bindestrich (-)
- Rest muss codiert werden
  - nicht leserlich
  - fehleranfällig bei Client-Implementierung

## Fehlerbehandlung

### HTTP-Statuscodes

- HTTP bringt über 70 definierte Status-Codes mit
- Codes mit 4 oder 5 beginnend sind Fehler-Codes
- naheliegendsten Fehlercode verwenden
- Anti-Pattern: proprietäre Fehlercodes über 2XX tunneln

### Proprietäre Fehlercodes

- häufig langen die Standard-Codes nicht aus
- Standardcodes im Rumpf um proprietäre Codes erweitern
- Beispiel: twilio

```
{
  "status": 400,
  "message": "No to number is specified",
  "code": 21201,
  "more_info": http://www.twilio.com/docs/errors/21201
}
```

#### NOTE

Bei Twilio (einem Anbieter u.a. zum SMS-Versand/-Empfang) ist besonders hervorzuheben, dass ein Link mit weiterführenden Informationen mitgeschickt wird.

So hat man seine eigene Dokumentation gleich mitintegriert.

- Beispiel: GitHub

```
HTTP/1.1 422 Unprocessable Entity
Content-Length: 149
{
  "message": "Validation Failed",
  "errors": [{
    "resource": "Issue",
    "field": "title",
    "code": "missing_field"
  }]
}
```

#### NOTE

GitHub liefert hier detaillierte Informationen zu dem allgemeinen Status-Code 422.

Man kann leicht erkennen, dass das Feld **title** nicht mitgeschickt wurde.

## Versionierung

### Query-Parameter

- Beispiel: <http://api.example.com/products?version=v2>
- Spezialfall: Datum <http://api.example.com/orders?version=20160501>
- (+) einfach zu verstehen
- (-) widerspricht dem Sinn von Query-Parametern

#### NOTE

Bei dem Datum-Ansatz gibt der Client das Datum an, bei dem er angefangen hat, die API zu nutzen.

Der Produzent entscheidet dann aufgrund des Datums, welche Version genommen wird.

## Header

- proprietär
  - **api-version: v1**
  - 'X'-Präfix soll nicht mehr verwendet werden (dies war lange Zeit die Empfehlung für eigene Header)
- **Accept**
  - **application/vnd.example.v2.orders+json**

## URI

```
http://api.example.com/[version]/orders
http://api.example.com/orders/[version]
```

- (o) bei erster Variante werden alle Ressourcen versioniert
  - evt. müssen so Clients geändert werden, obwohl sie auch so funktionieren könnten
- (+) Ansatz ist leicht verständlich
- (+) leicht zu testen
- (-) widerspricht dem URI-Ansatz
  - URI = eindeutige Identifikation einer Ressource, nicht deren Version

# Partielle Rückgabe

## Grundlage

- häufig macht es keinen Sinn, große Listen am Stück zurück zugeben
  - z.B. Treffer einer Anfrage bei einer Suchmaschine
- einfache Begrenzung: `/api/search?q=foobar&max-results=10`
- Problem: nächsten 10 Treffer, würden die ersten 10 wieder mitübertragen
  - einfache Begrenzung: `/api/search?q=foobar&max-results=20`

## Paging

- Inhalt / Ergebnis aufteilen in Blöcke
- Blöcke abfragen
- Beispiel Log-Abfrage: `/api/logs?page=2&count=1000`
- Beispiel Google: `https://www.google.com/search?q=test+dhbw&start=10`
  - Google-Suche hat feste 10er Blöcke und `start`-Parameter
  - schlecht, weil `start=11` funktioniert, aber komische Ergebnisse liefert

## Cursoring

- `cursor` als Parameter
- `api/big-list?cursor=-1`
  - Response: u.a. `{next: 83621, previous=-1}`
  - mit `next` (⇒ `api/big-list?cursor=83621`) zum nächsten Abschnitt
  - mit `previous` zum vorherigen Abschnitt
  - `next: 0` ⇒ Ende

# Feldangabe

- benötigte Felder als Parameter übertragen
- ungewollte Felder werden ausgeschlossen
- `/orders/order- 42?fields=creation_date,positions(count,article/name)`
  - nur die Felder `creation_date` und `positions` werden abgefragt
  - `positions` ist zusätzlich auf `count` und `article` eingeschränkt
  - `article` ist auf `name` eingeschränkt

# GraphQL

- JSON-basierte Query-Language Spezifikation von Facebook
- sehr mächtig und flexibel
- zahlreiche Server- und Client-Libraries (u.a. Java, C#, ...)

Beispiel-Anfrage	Beispiel-Antwort
<pre>{   article {     id     name   } }</pre>	<pre>{   "article": {     "id": "42",     "name": "HDD 8 TB"   } }</pre>

# Sicherheit

## Standardverfahren

- alle Standardverfahren nutzen `Authorization`-Header
- HTTP Basic Authentication
  - `Authorization: Basic bmFtZTpwYXNzd29yZA==`
  - `{Benutzername}:{Passwort}` in Base64
- OAuth
  - Services agieren im Namen des Nutzers
  - `Authorization: Bearer {AccessToken}`
- OpenID Connect
  - `Authorization: Bearer {JsonWebToken}`
  - im `JsonWebToken` sind Benutzerrechte codiert

- basiert auf OAuth **AccessToken**
- ersetzt Cookies

# Literatur

*API-Design* von Kai Spichale

