Softwaretechnik 2

Vorlesung Informatik Bachelor, SS 2021

Prof. Dr.-Ing. Stefan Bente

www.archi-lab.io

Script zur Veranstaltung

in Form von kommentierten Folien

https://www.youtube.com/channel/UC29euiLjp5m-hPoU3QP3nGA

Siehe auch unser ArchiLab-Youtube-Channel mit Videos zur Veranstaltung. Zu jedem Inhaltsblock in diesem Script gibt es ein Video.

Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=283EGmUxRN0

Warum empfehle ich Ihnen nicht "ein Buch" für diese Veranstaltung?

Teil 1

Gute Bücher, die man nicht unbedingt besitzen muss

- Es gibt leider nicht **das** gute Software-Engineering-Buch ☹
- Folgende Buch-Typen findet man: (abgewandelt nach nach Siedersleben, J. (2004). *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar* (1st ed.). dpunkt, S. 12)

1. **Softwaretechnik-Klassiker**: große, umfangreiche „Standard-Lehrwerke“
2. **Theorie-Werke** aus der universitären Forschung, ohne viel Rückhalt in der Praxis
3. Bücher zu **Patterns** / Entwurfsmustern / Architekturstilen
4. **Technik-Bücher** (z.B. zu JEE, AngularJS, …)
5. **gute Ratschläge** (Ratgeber zu bestimmten Detailthemen)

1) Softwaretechnik-Klassiker

- Große, umfassende “Standardwerke”, die alle Aspekte des SW-Lebenszyklus abdecken. I.d.R. sehr umfangreich (800+ Seiten).
- Oft recht prozess- und standardlastig. Hilft eher nicht bei der Frage “Was tue ich als erstes, zweites, drittes, wenn ich ein SW-System neu erstellen will”.
- Gut geeignet als Nachschlagewerk – jedenfalls, wenn man in großen Organisationen unterwegs ist und bestimmte Standards beim Vorgehen und der Dokumentation einhalten muss. Dann kann man dort gut einmal nachschlagen, um den Aspekt besser verstehen und einordnen zu können.
- **Folgende Beispiele gibt es hier zu nennen (kein Anspruch auf Vollständigkeit!)**

- Klassische Standardwerke zu Software Engineering im Ganzen
  - Balzert, Helmut. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. 2012. Heidelberg: Spektrum Akademischer Verlag, 2011. (Es gibt mehrere Bände aus dieser Reihe)
  - Sommerville, I. (2016). *Software Engineering* (10. Auflage). Pearson Studium.
  - Booch, Grady, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*. 3rd ed. The Addison-Wesley Object Technology Series. Upper Saddle River, NJ: Addison-Wesley, 2007.
  - Jackson, Michael. *Problem Frames & Methods: Analysing and Structuring Software Development Problems*. Harlow: Addison-Wesley Longman, Amsterdam, 2000.

1) Softwaretechnik-Klassiker (Forts.)

- Empfehlenswerte Standardwerke speziell zu Architekturthemen (nur Beispiele)
  - Capgemini. (2015). Architecture Guidelines for Application Design v2.0.
  - Dustdar, Schahram, Harald Gall, and Manfred Hauswirth. *Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software*. 1st ed. Springer Berlin Heidelberg, 2003.
  - Gregor Engels, Andreas Hess, Bernhard Humm, Oliver Juwig, Marc Lohmann, Jan-Peter Richter, Markus Voß, and Johannes Willkomm. *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. 1., Aufl. dpunkt.verlag, 2015.
  - Lilienthal, Carola. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. 1st ed. dpunkt.verlag GmbH, 2015.
  - Newman, S. (2015). *Building Microservices* (1st ed.). O'Reilly and Associates.
  - Siedersleben, Johannes. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. 1st ed. Heidelberg: dpunkt, 2004.
  - Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen*. 1., Auflage. dpunkt.verlag, 2015.

- Fokus auf Modellierung mit UML
  - Als Referenz sollten Sie immer in den frei verfügbaren Standard schauen:
    - OMG. "OMG Unified Modeling Language, Version 2.5.1," 2015. https://www.omg.org/spec/UML/2.5.1/PDF
  - Darüber hinaus gibt es eine Vielzahl von Büchern dazu (keine Empfehlung von mir an dieser Stelle)

2) Ergebnisse der akademischen Forschung

?

- In der Softwareentwicklung scheinen alle nennenswerten Innovationen der letzten zwei Jahrzehnte **NICHT** aus der universitären Forschung gekommen zu sein.
  - Das ist schon ungewöhnlich – und bei Medizin, Chemie, Maschinenbau, … etc. anders. Auch in anderen Bereichen der IT (z.B. KI, Data Science) gilt das sicher nicht.
  - In der Softwareentwicklung ist eher die Community von anerkannten Expert*innen der Innovationstreiber.
  - Daher keine Bücher hier aufgeführt!

- **Einzige Ausnahme, die mir einfällt:** Dissertation von Roy Fielding, die die Grundlage von REST darstellt
  - Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* [University of California, Irvine]. http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf

3) Bücher zu Patterns / Entwurfsmustern / Architekturstilen

- Patterns haben sich als „Stilelement“ im Design von Software durchgesetzt. Es gibt eine Menge guter Bücher dazu. Man findet allerdings auch viele sinnvolle Informationen in Onlinequellen.
- Der Klassiker ist das „Gang of Four“ Buch:
  - Gamma, Erich, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* 1st ed., Reprint. Reading, Mass: Prentice Hall, 1994.

- Weitere empfehlenswerte Werke dazu:
  - Eilebrecht, Karl, and Gernot Starke. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 4th ed. Berlin: Springer Vieweg, 2013.
  - Goll, Joachim. *Architektur- und Entwurfsmuster der Softwaretechnik*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014. http://link.springer.com/10.1007/978-3-658-05532-5.
  - Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 01 ed. Boston: Addison Wesley, 2003.
    - Auch als frei verfüglicher, sehr empfehlenswerter Blog, der die wesentlichen Informationen beinhaltet: http://www.enterpriseintegrationpatterns.com/index.html.

4) Technik-Bücher (z.B. zu JEE, AngularJS, …)

- Hier gebe ich Ihnen keine Empfehlungen – schauen Sie selbst, was Sie brauchen. Der Markt ist viel riesig, und in dieser Veranstaltung geht es nicht um eine bestimmte Technologie (auch wenn wir Technologien vorgeben).
- Dieser Typ von Buch veraltet i.a. schnell. Man wird so ein Buch, wenn man es sich kauft oder aus der Bibliothek ausleiht, vermutlich einmal mehr oder weniger komplett lesen, später aber höchstens noch als Referenz nutzen.
  - Da gibt es dann gute Onlinequellen (z.B. Stackoverflow) als Konkurrenz.

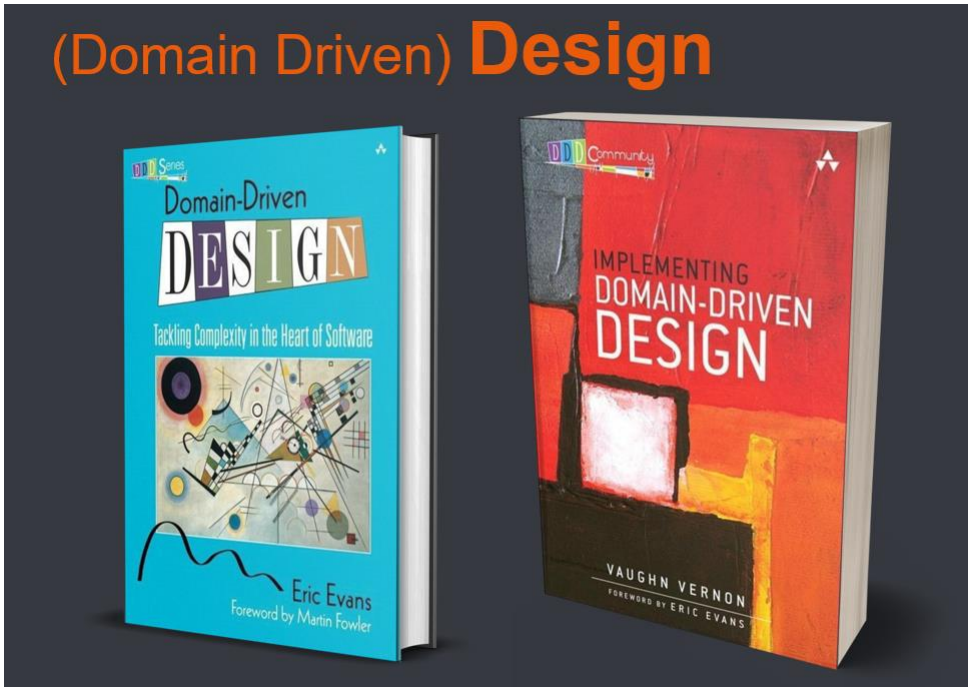
5) gute Ratschläge (Ratgeber zu bestimmten Detailthemen)

- “Ratgeber” sind Bücher zu bestimmten Aspekten des Entwicklungsprozesses.
- Meist liest man sie von vorn bis hinten (vielleicht unter Auslassung bestimmter Kapitel), weil der im Ratgeber behandelte Aspekt gerade von besonderer Bedeutung ist.
- Solche Bücher kann man später immer wieder als Nachschlagewerke nutzen.

- Beispiele für gute Ratgeber
  - Hruschka, P., & Starke, G. (2014). *Knigge für Softwarearchitekten—Reloaded*. entwickler.press.
  - Toth, Stefan. *Vorgehensmuster für Softwarearchitektur: Kombinierbare Praktiken in Zeiten von Agile und Lean*. 2., aktualisierte und erweiterte Auflage. München: Carl Hanser Verlag GmbH & Co. KG, 2015.
  - Zörner, Stefan. *Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten*. 2., Aufl. München: Carl Hanser Verlag, 2015.
- Für ST2 relevant: REST-Ratgeber
  - Massé, M. (2011). *REST API Design Rulebook* (1. Aufl.). Beijing: O’Reilly and Associates.
  - Tilkov, Stefan, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web, p. 11ff. 3. Aufl. Heidelberg: dpunkt.verlag GmbH, 2015.

- Während die vorgenannten Arten von Büchern meist doch nur punktuell benötigt werden, sind die nachfolgenden vier Bücher (nach Meinung die Autors!) von einer bleibenden Qualität – man kann sich dort immer wieder Anregungen holen.
- Für die Veranstaltung ST2 werden diese vier Bücher eine größere Rolle spielen.

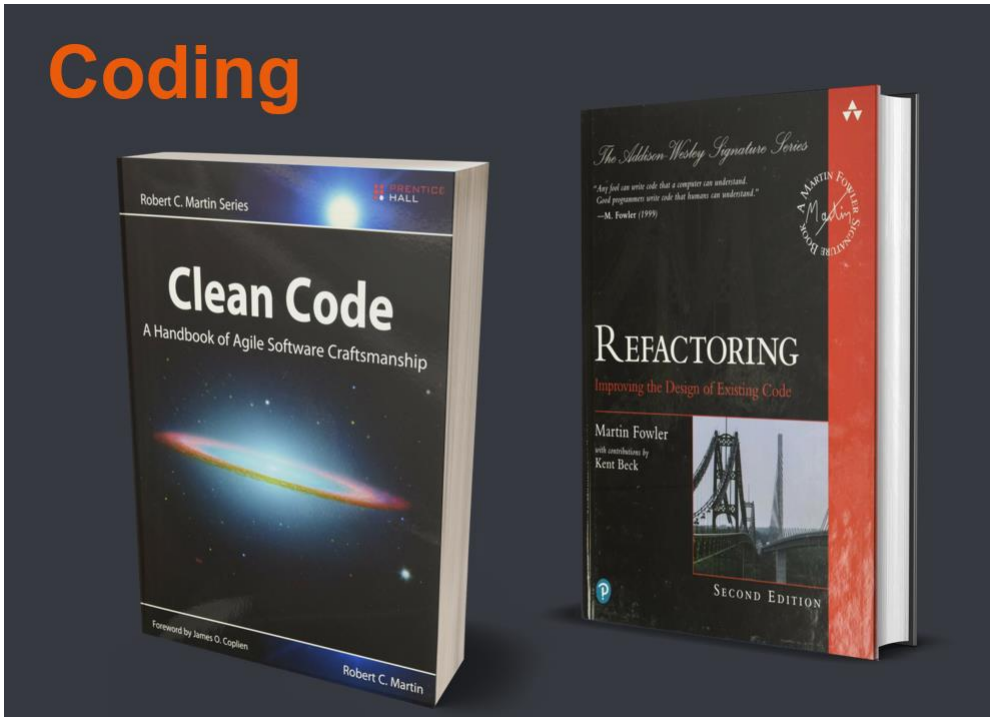
Domain-Driven Design



- DDD wurde in dieser Veranstaltung bereits ausführlich eingeführt.
- Mit diesen beiden Büchern hat man eine umfassende Grundlage zu diesem Designansatz.
- Evans (“blaues Buch”) ist gut für die Grundlagen, aber etwas sperrig zu lesen.
- Vernon (“rotes Buch”) liest sich flüssiger und ist gut als aufbauende Fortsetzung zu Evans zu lesen.

- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software* (1 edition). Addison-Wesley Professional.
- Vernon, V. (2013). *Implementing Domain-Driven Design* (1 ed.). Addison Wesley.

Coding

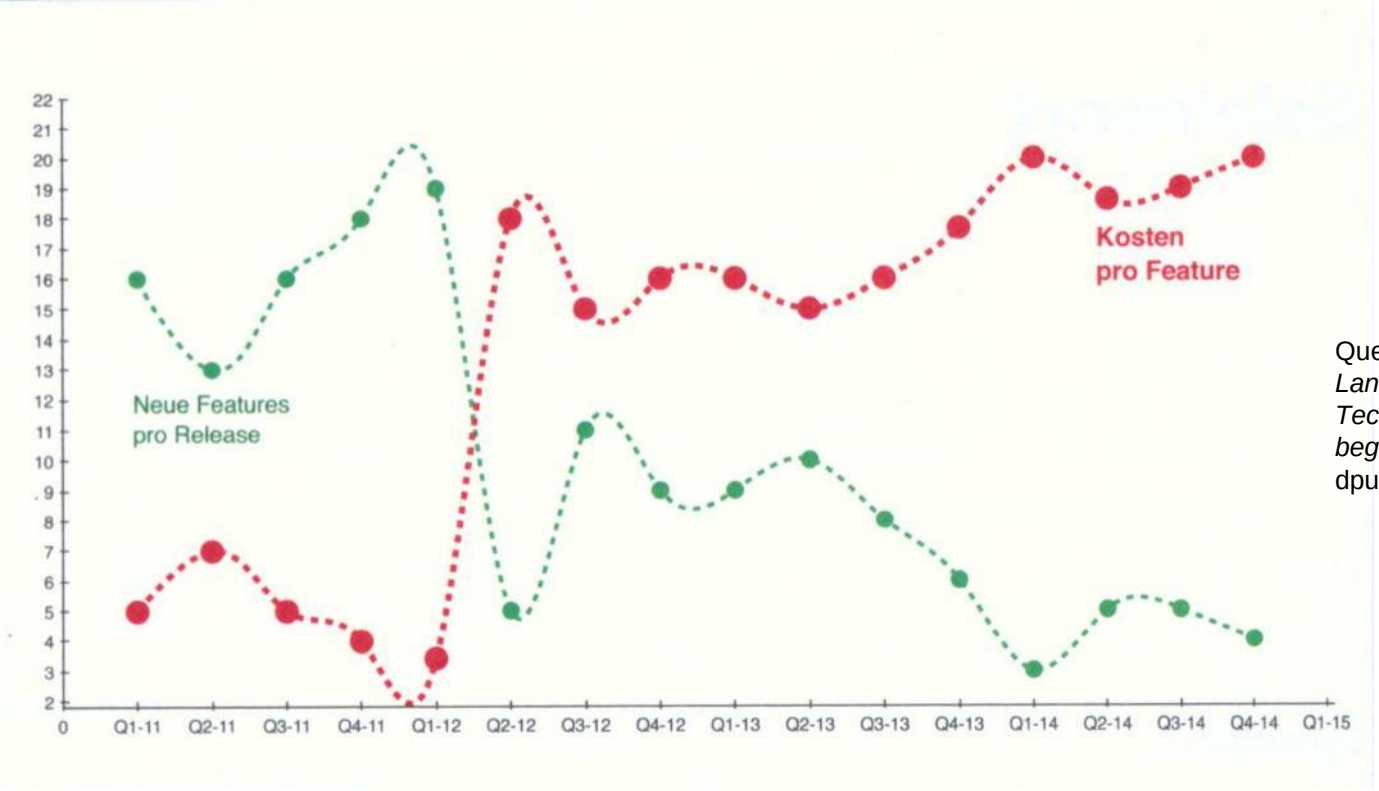


- Bob Martin (a.k.a. “Uncle Bob”) hat diesen modernen Klassiker als Beschreibung einer “handwerklich sauberen” Haltung beim Coden geschrieben.
- Martin Fowlers Buch beschreibt (in gewisser Weise aufbauend darauf), wie man technische Schuld (schlechten Code, schlechte Architektur) Stück für Stück besser macht.
- Die Inhalte beider Bücher werden in der ersten Phase von ST2 (Meilenstein M1) angewendet.

- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.

https://www.youtube.com/watch?v=QHNMa4qCGJc

Viele SW-Systeme entwickeln sich so – wieso?



Quelle: Lilienthal, Carola. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. 1st ed. dpunkt.verlag GmbH, 2015.

- Nach der initialen Implementierung der ersten Software-Version werden die Aufwände für neue Features mit der Zeit immer höher.
- Dadurch kann dasselbe Team weniger Features implementieren (grüne Linie), das Innovationstempo nimmt also ab.
- Die Features werden dadurch immer teurer – die Produktivität sinkt. Konkurrierende Softwareprodukte bekommen einen Kostenvorteil.

- Woran liegt das?

Lösung: Technische Schulden regelmäßig abtragen!

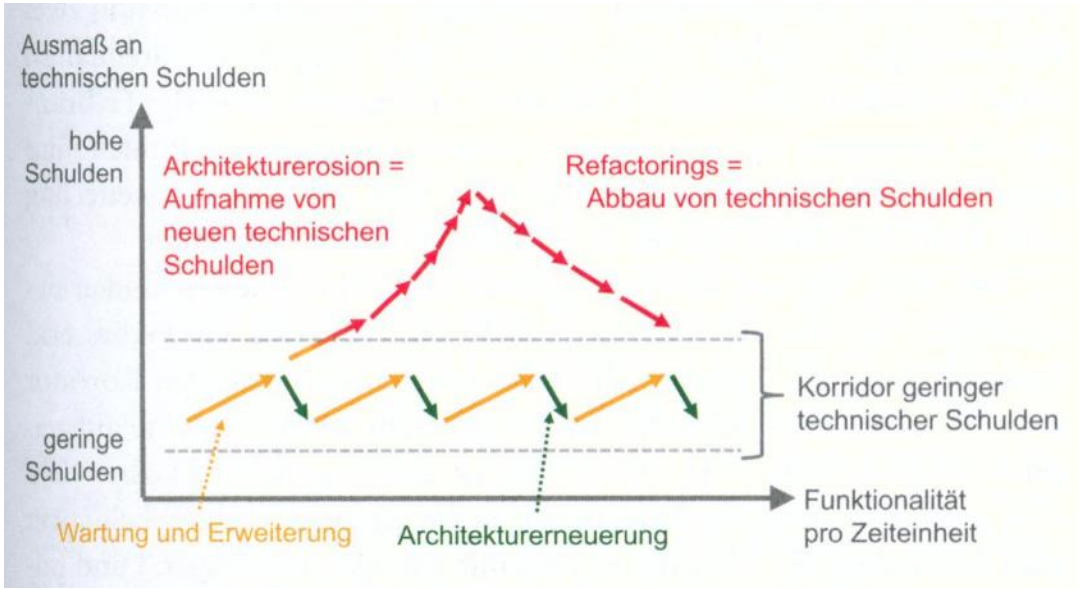
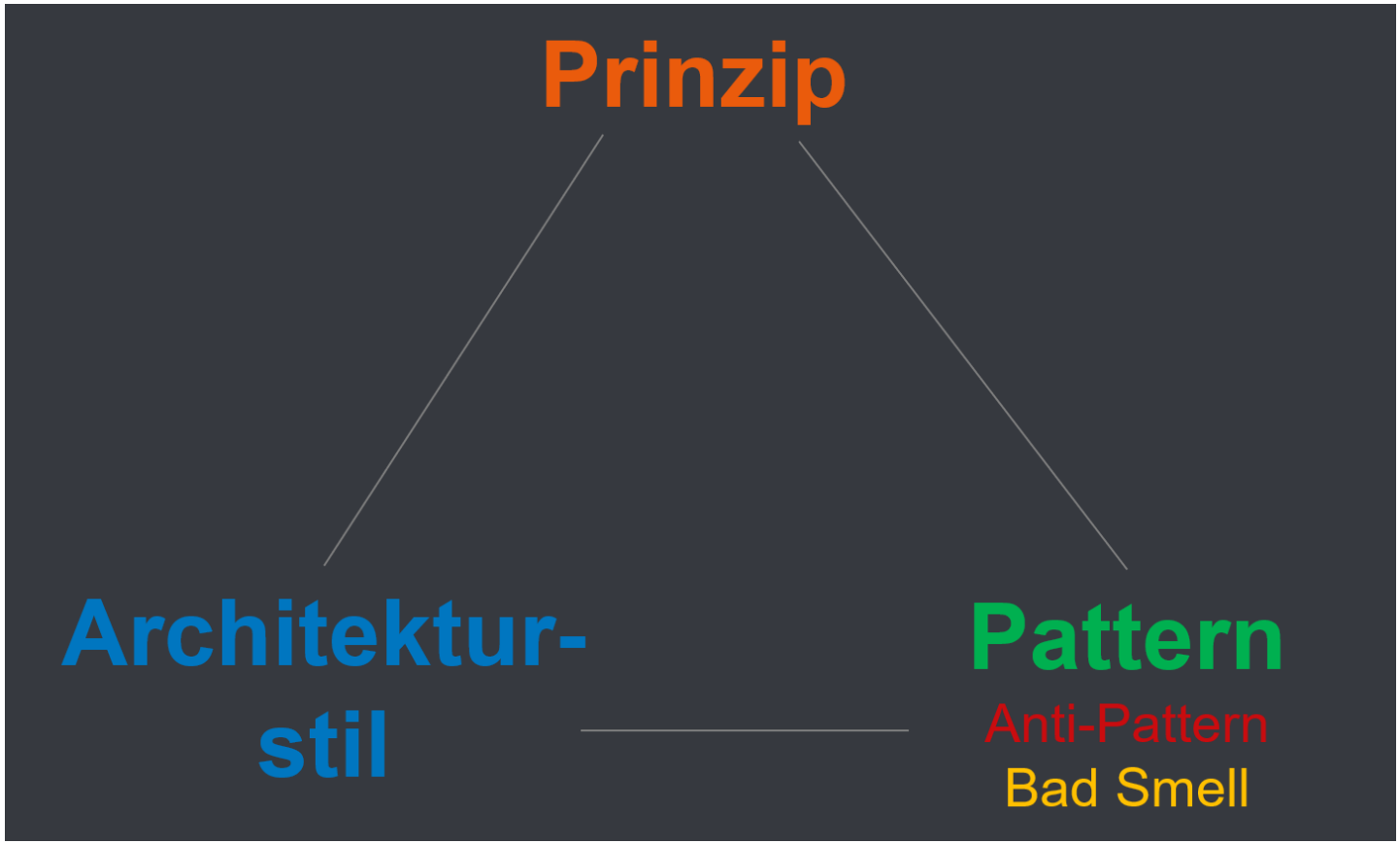


Bild: [Lilienthal], S. 6

- Ein wesentlicher Grund hierfür ist die sogenannte “technische Schuld”. Diese entsteht, indem nach und nach Konventionen nicht mehr eingehalten werden, die Klarheit eines Software-Designs verloren geht, ein gewisser “Wildwuchs” stattfindet.
- Dieser Prozess ist nicht vermeidbar. Es kommen neue Entwickler*innen und Teams dazu, die die alten Konventionen nicht alle kennen. Man verwendet neue Technologien, die man erst nach und nach lernt zu beherrschen.
- Wichtig ist daher, diese technische Schuld regelmäßig zu analysieren und zu beheben (gelb-grüne “Sägezahnlinie” unten). Wenn man es zu lange laufenlässt, entsteht die rote Kurve. Irgendwann ist die technische Schuld so groß, dass man man die Software fast komplett neu schreiben muss.

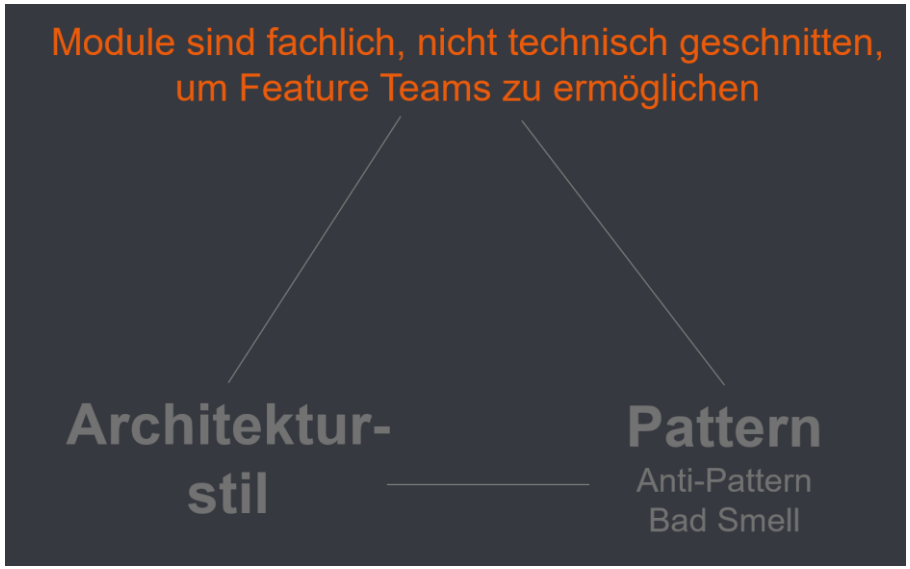
- **Beim Einhalten des Korridors geringer technischer Schuld (unten) helfen Prinzipien, Architekturstile und Pattern.**

„Wirkungs-Dreieck“: Prinzipien, Patterns, Architekturstile



- Prinzipien, Patterns und Architekturstile bilden ein „Wirkungs-Dreieck“: Sie helfen und bedingen einander.

Prinzipien



- Ein Prinzip …
  - gilt immer
  - gilt überall
  - beschreibt allgemeingültige “Leitplanken”
  - ist i.d.R. abgeleitet aus dem Geschäftsmodell (und anderen nicht-technischen Randbedingungen)

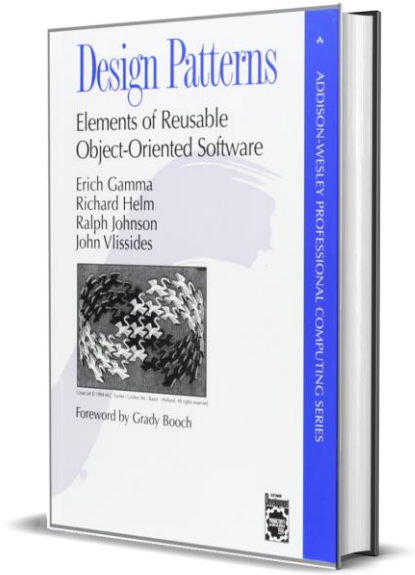
- Ein Beispiel ist “**Module sind fachlich, nicht technisch geschnitten, um Feature Teams zu ermöglichen**”
- Nehmen wir an, unsere Firma ist ein eCommerce-Portal. Wir sind extrem darauf angewiesen, dass neue Features **ganz schnell** ins System kommen.
  - Beispiel: Es ist Ostern, und wir konnten günstig eine Charge chinesischer Plüschhasen ersteigern. Dann brauchen wir ganz schnell eine Kampagne auf unserer Seite, die bei Bestellungen über 100 € einen kostenlosen Plüschhasen verspricht und den dann auch der Bestellung beilegt.
  - Das geht nur, wenn unsere Teams sich nicht groß absprechen müssen, sondern für ein Feature – z.B. “Warenkorb” – komplett verantwortlich sind, also vom UI über die Datenhaltung bis zum Hosting.
  - Hier wiederum setzt unser Prinzip an: Wir haben also Teams für “Warenkorb”, “Bezahlung”, “Versand” – und keine Teams für “UI”, “Database”, “Betrieb” etc.
- **Dieses Prinzip ergibt sich also direkt aus geschäftlichen Notwendigkeiten**.

Architekturstil

- Architekturstil = Softwarestruktur “im Großen”
- Element = Architekturbaustein
  - Komponente
  - Schnittstelle
  - Klasse
  - …
- Der Stil legt fest, welche Elemente es gibt, und wie sie zusammenhängen.

- Nehmen wir das Beispiel von eben. Unser Prinzip ist „ **Module sind fachlich, nicht technisch geschnitten, um Feature Teams zu ermöglichen.**”
- Dann kann man sich fragen: Wie muss meine Software strukturiert sein, um dieses Prinzip umsetzen zu können?
- Die Antwort darauf ist ein Architekturstil. In diesem Fall ist eine **Schichtenarchitektur mit Domain- und Application Layer nach DDD-Konzept** der passende Architekturstil.
- Dieses Stil haben wir teilweise in ST1 schon kennengelernt. Man denkt ein System vom fachlichen Schnitt (Warenkorb, Bezahlung, Versand, …) und isoliert die fachlichen Module so weit wie möglich. Dafür nimmt man bewusst Redundanzen in Kauf – das ist der Kern dieses Architekturstils.

Pattern



- Pattern = Wiederverwendbares Lösungsmuster
  - Ursprung: Christopher Alexander (1977) im Bauwesen
- Ende 1980er auf SW übertragen („Gang of Four“-Buch von Gamma et al., siehe rechts)
- Pattern können vieles betreffen: Code, Daten, Modul-/Dateistrukturen, Namen, …

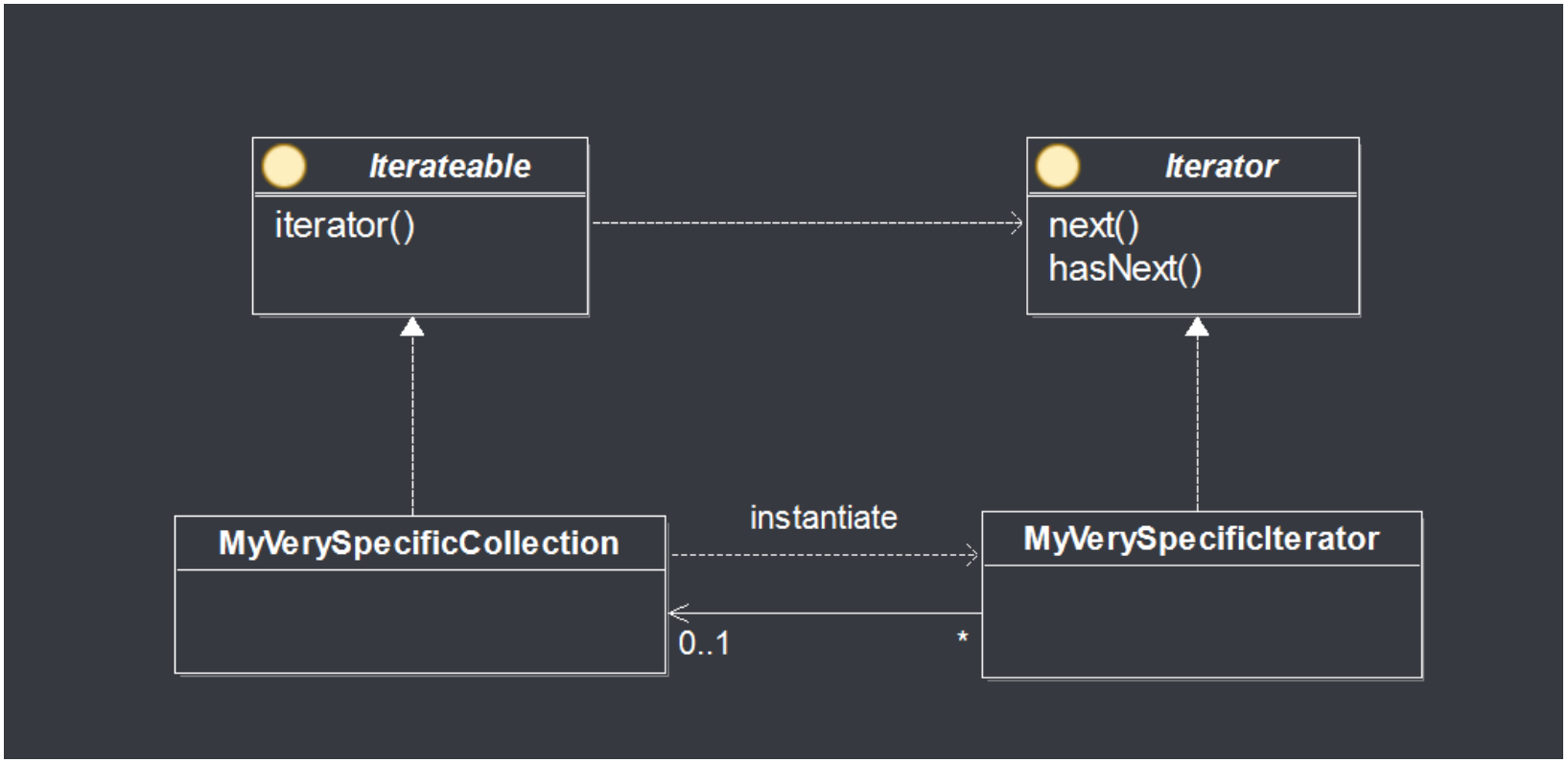
- In diesem Sinne kann man sagen, dass **ein Architekturstil die Summe seiner Patterns ist**, also durch Patterns umgesetzt wird.

- "Das Hauptziel von Mustern für den Softwareentwurf ist es,
  - einmal gewonnene Erkenntnisse **wiederverwendbar** zu machen
  - und durch ihre Anwendung die **Flexibilität** einer Architektur zu erhöhen.“ [Goll], S. 67

- "Muster haben das Ziel,
  - die Eigenschaften von Architekturen zu verbessern.
  - Hierzu gehören beispielsweise die folgenden Eigenschaften:
  - **Verständlichkeit** und
  - **Erweiterbarkeit**.“ [Goll], S. 65

- (Zitate aus Goll, Joachim. *Architektur- und Entwurfsmuster der Softwaretechnik*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014.

Beispiel für ein allgemein anwendbares Pattern: Iterator



- “Iterator” ist ein allgemein anwendbares Pattern in der Softwareentwicklung, um in immer gleicher Weise über spezifische Collections iterieren zu können.

Beispiel für Architekturstil: Repräsentationsbau

Bild: Felix Lühning, http://www.amalienburg-gottorf.de/pageID_5167788.html

Bau-Stil

- Repräsentatives gebäude
  - kein Zweckbau
- Optik wichtig
  - Symmetrischer Aufbau
  - stimmige Proportionen
- Innen: Zweiteilung
  - Repräsentations-bereich
  - Privatbereich

Architekturstil = Summe der (Konstruktions-)Patterns

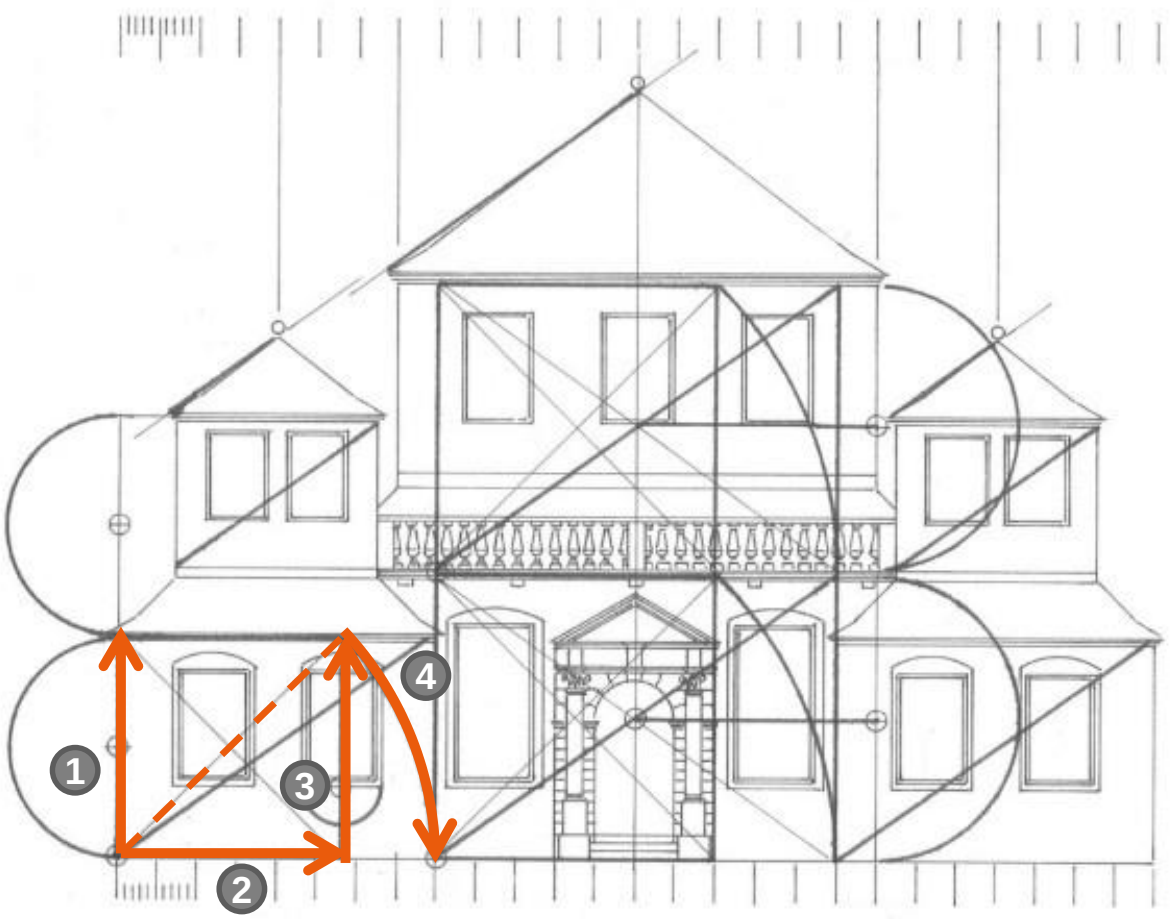


Bild und Quelle: Felix Lühning, http://www.amalienburg-gottorf.de/pageID_5167748.html

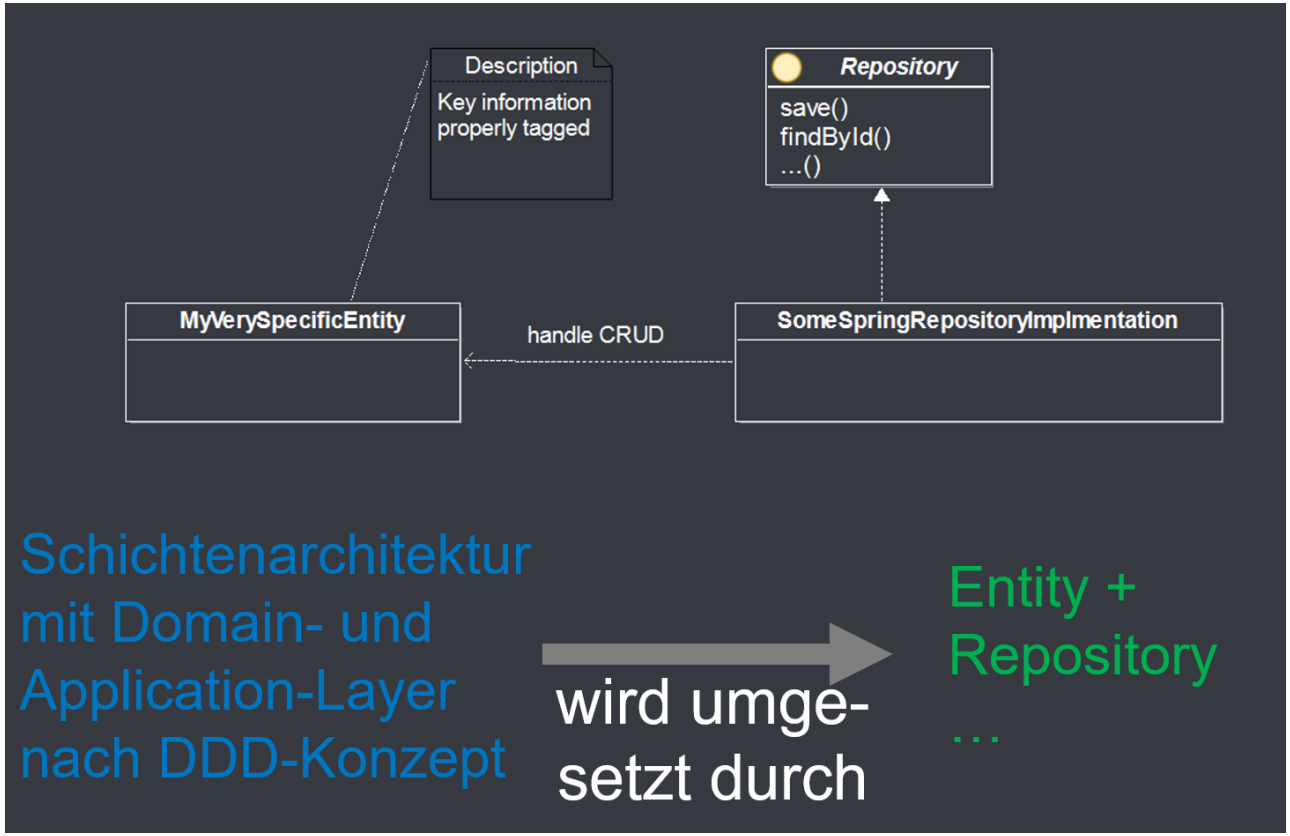
Elemente

- Gebäude-Block
- Dachform 1
- Dachform 2

Element-Regeln

- Höhe zu Breite = 1 zu $\sqrt{2}$
- => stimmige Proportionen, leicht zu konstruieren

Umsetzung DDD-Schichtenarchitektur durch Entity + Repo



- Führen wir noch einmal das Beispiel von eben weiter …
  - Prinzip: „ **Module sind fachlich, nicht technisch geschnitten, um Feature Teams zu ermöglichen.**”
  - Architekturstil: **Schichtenarchitektur mit Domain- und Application Layer nach DDD-Konzept**
- … dann ist „**Entity + Repository**“ ein Pattern, das dabei hilft, diesen Architekturstil umzusetzen.

Patterns, Anti-Patterns, Bad Smells

- Anti-Pattern
  - Häufig anzutreffender, schlecht geeigneter Ansatz zur Lösung eines typischen Problems
  - Oft Ergebnis fehlender oder missverstandener Architektur

- Bad Smell
  - Symptom (Architektur, Code) für tieferliegendes Problem
  - kann u.U. gerechtfertigt sein – in der Regel aber nicht

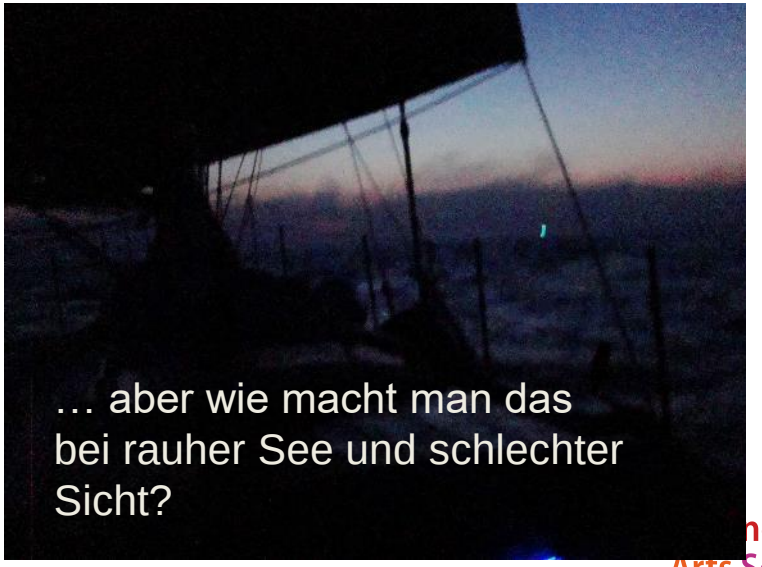
- (Design) Pattern
  - Wiederverwendbares Lösungsmuster
  - *Ursprung:* Christopher Alexander (1977), Konzept einer „Pattern Language“ für Bauten (*)
  - Ende 1980er auf SW-Entwicklung übertragen (**)
  - Vollständig akzeptiertes Konzept in der SW-Architektur (***)

Quellen:
(*) C. Alexander et al.: A Pattern Language: Towns, Buildings, Construction (1977). Oxford University Press
(**) Beck, Kent; Ward Cunningham (1987). "Using Pattern Languages for Object-Oriented Program". OOPSLA '87.
(***) Gamma, E. et al.: Design Patterns. Elements of Reusable Object-Oriented Software. 1st ed., Reprint. Reading, Mass: Prentice Hall, 1994.

Patterns, Anti-Patterns und Bad Smells helfen beim Steuern in Richtung eines Prinzips!

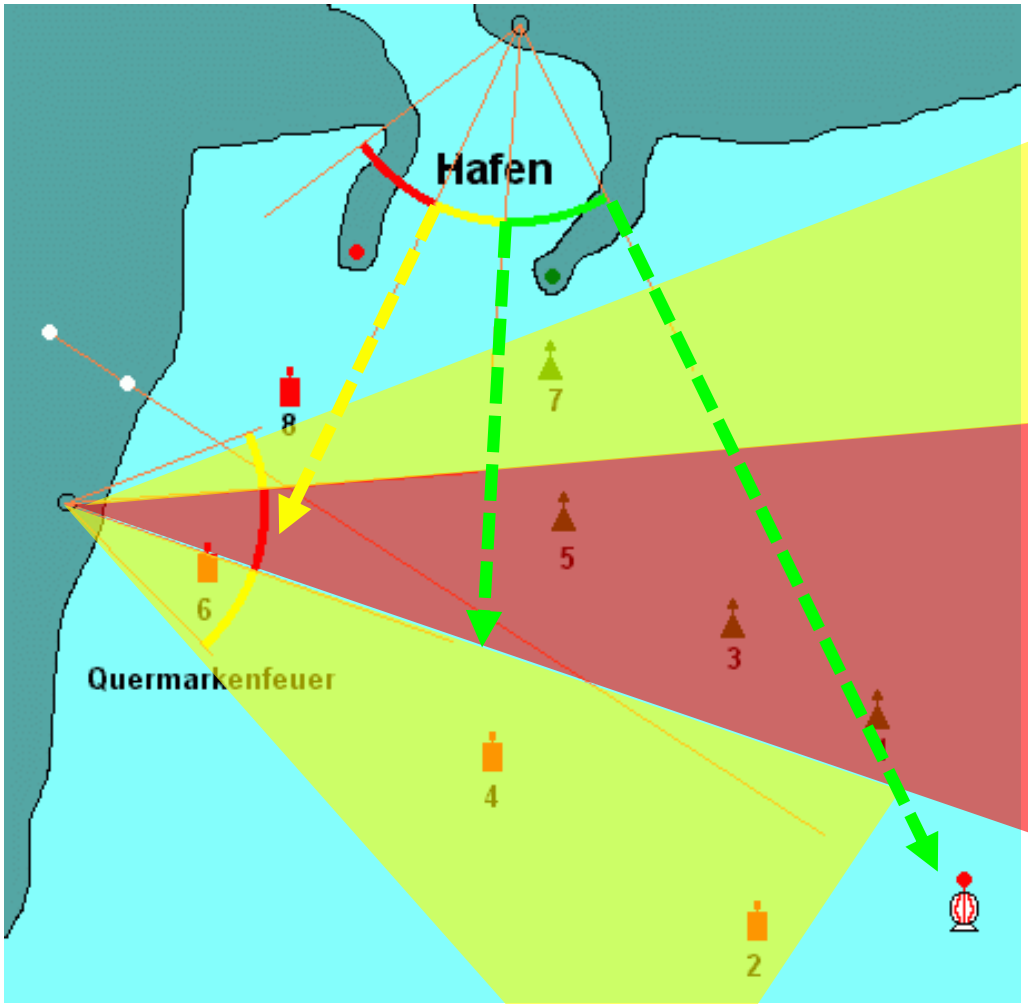
- Wir schauen uns das wieder anhand eines Bildes an. Nehmen wir an, das Prinzip ist unser “Leuchtturm”, auf den wir zusteuern beim Implementieren der Software …

Prinzipien = Leuchttürme …



- Softwareentwicklung ist manchmal so, als wäre man bei schlechtem Wetter und Dunkelheit in einem kleinen Boot auf dem Wasser unterwegs.
- Man sieht kaum die Hand vor Augen, man ist sich nicht sicher, ob man alles richtig macht …
- Vielleicht ist man schon längst ab vom Kurs, und steuert auf die Küste oder ein Unterwasser-Hindernis zu …?
- Patterns, Anti-Patterns und Bad Smells sagen einem, ob man noch auf Kurs ist. Um in dem Seefahrts-Bild zu bleiben: sie entscheiden, in welcher Farbe man den Leuchtturm sieht.

Lösung in der Nautik: Leuchtfeuer / Sektorenfeuer

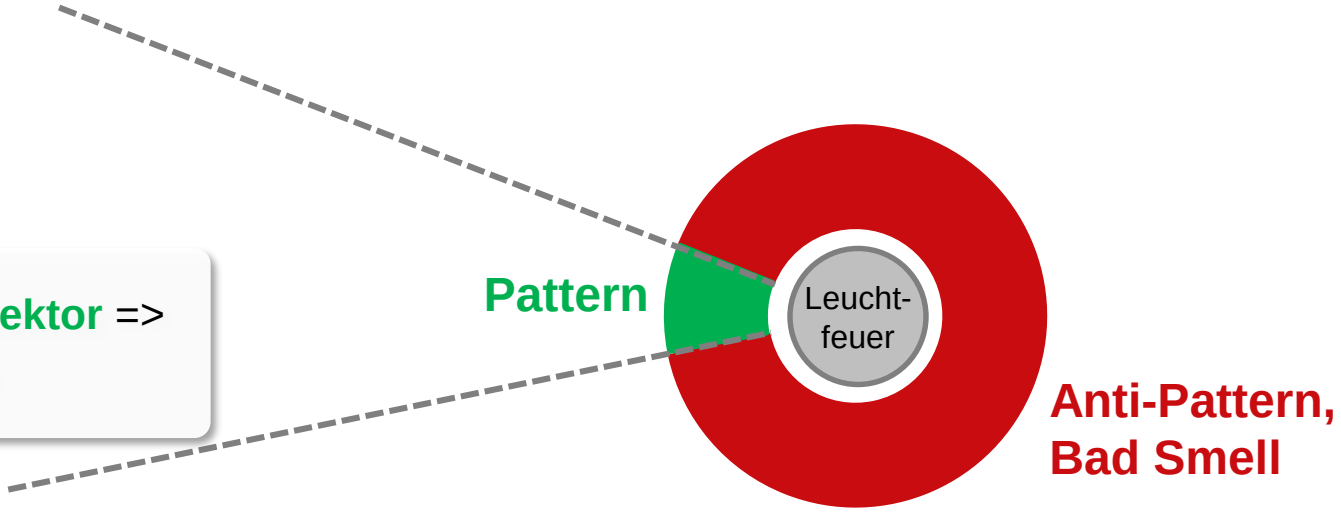


In diesem Sektor:
Leuchtfeuer erscheint **GELB**

In diesem Sektor:
Leuchtfeuer erscheint **ROT**

Prinzipien, (Anti-)Patterns & Bad Smells = zusammen …

- Ich sehe den **grünen Sektor** =>
- mein Schiff ist auf Kurs



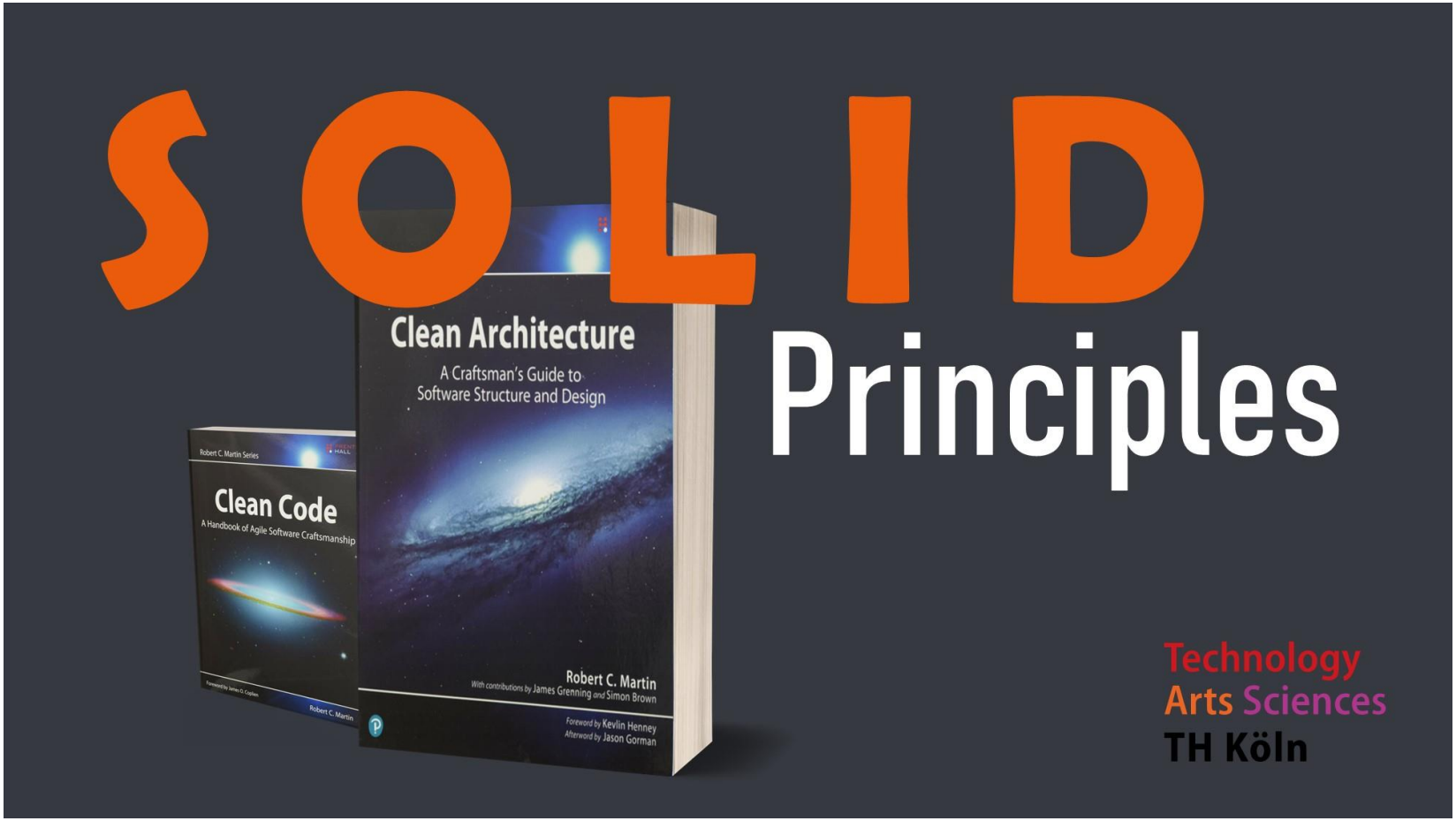
- Ich sehe den **roten Sektor** =>
- falscher Kurs!

Architekturprinzip

- Prinzipien sind wie (Sektoren-)Leuchtfeuer,
- Patterns und Anti-Patterns wie dessen Sektoren

Noch einmal im Ganzen …

- Damit ist das Dreieck vollständig beschrieben.
- Mit diesen drei Stilmitteln können Sie bessere Software entwickeln.
- In den nachfolgenden Videos lernen wir typische Prinzipien (**SOLID**) kennen und mit **Clean Code** eine Sammlung von hilfreichern Patterns und Anti-Patterns.

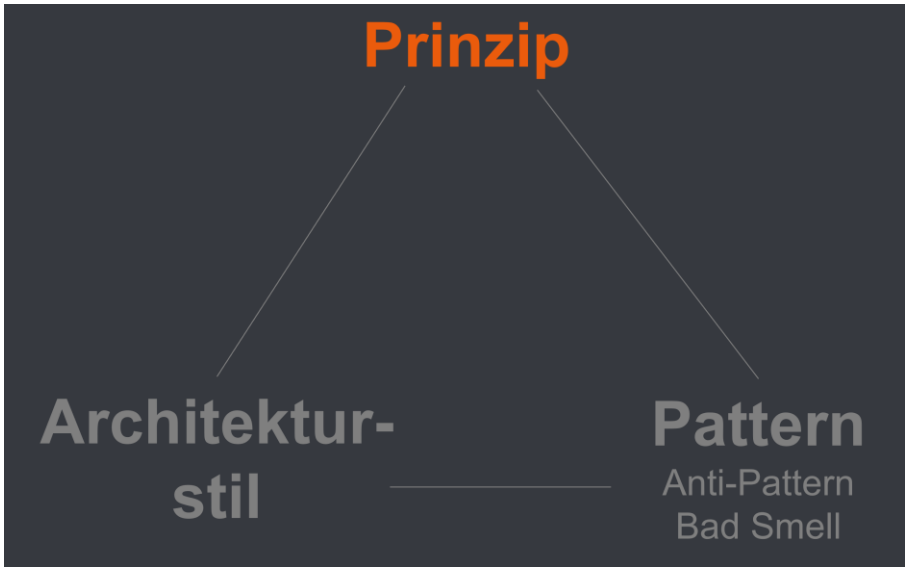


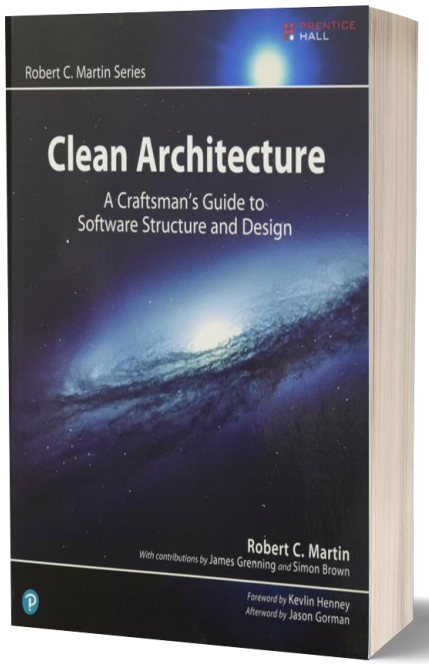
https://www.youtube.com/watch?v=kQDStoasH-Q

Was ist SOLID?

- The SOLID principles were conceptualized by Robert Martin in his paper “Design Principles and Design Patterns” (2000).
- SOLID is an acronym for five principles:
  - **S**ingle responsibility principle
  - **O**pen closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation princinple
  - **D**ependency inversion principle
- All of these principles focus on the structure and design of an application and it's source code.
- Each of these principles states a solution form common structural problems in a code base.
- The solutions are focused on the improvement of values:
  - flexibility
  - maintainability
  - extendability
  - simplicity

SOLID = Prinzipien

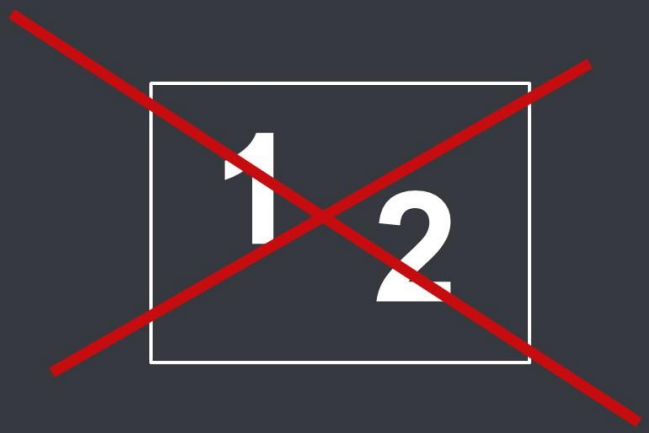




- In unserem „Dreieck“ aus Prinzipien, Architekturstil und Patterns fallen die SOLID-Prinzipien in die erste Kategorie.
- Sie sind Leitlinien auf einem relativ abstrakten Niveau, die immer gelten.
- Neben dem eben erwähnten Paper werden sie hauptsächlich in Martins Buch „Clean Architektur“ (Bild rechts) thematisiert.

Single Responsibility Principle

(a.k.a. Separation of Concerns)



Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=BFVLXYFlNXg

1) Eine Grillparty mit Freunden, und Sie kaufen für alle ein.

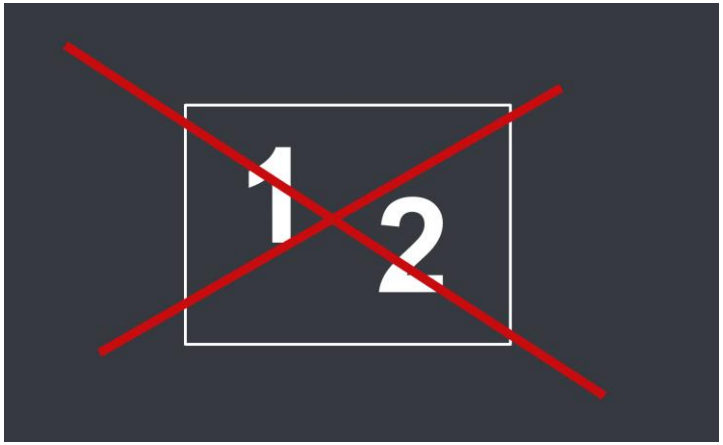
- Mehmet isst kein Schweinefleisch
- Hubbi ist Vegetarier
- Bogdan isst nur Bio
- Alina liebt Bratwurst
- Sabine mag nur diese kleinen marinierten Hühnchenfilets, die kriegt man beim Rewe in der Großmannstraße und wenn da nicht, dann auf jeden Fall beim Metzger Köhnmann in der Eifelstraße, das findest du ganz leicht, einfach am Neumarkt die erste links und dann

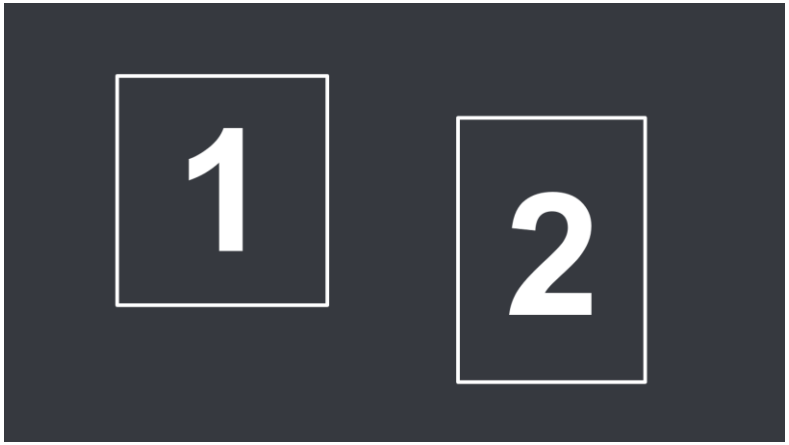
Eine Grillparty mit Freunden – was ist hier passiert?

- Getränke kaufen
- Grillgut für Mehmet besorgen
- Grillgut für Hubbi besorgen
- Grillgut für Bogdan besorgen
- Grillgut für Alina besorgen
- Grillgut für Sabine besorgen

Besser: Jeder besorgt sein eigenes Grillgut!

Single Responsibility Principle (a.k.a. Separation of Concerns)

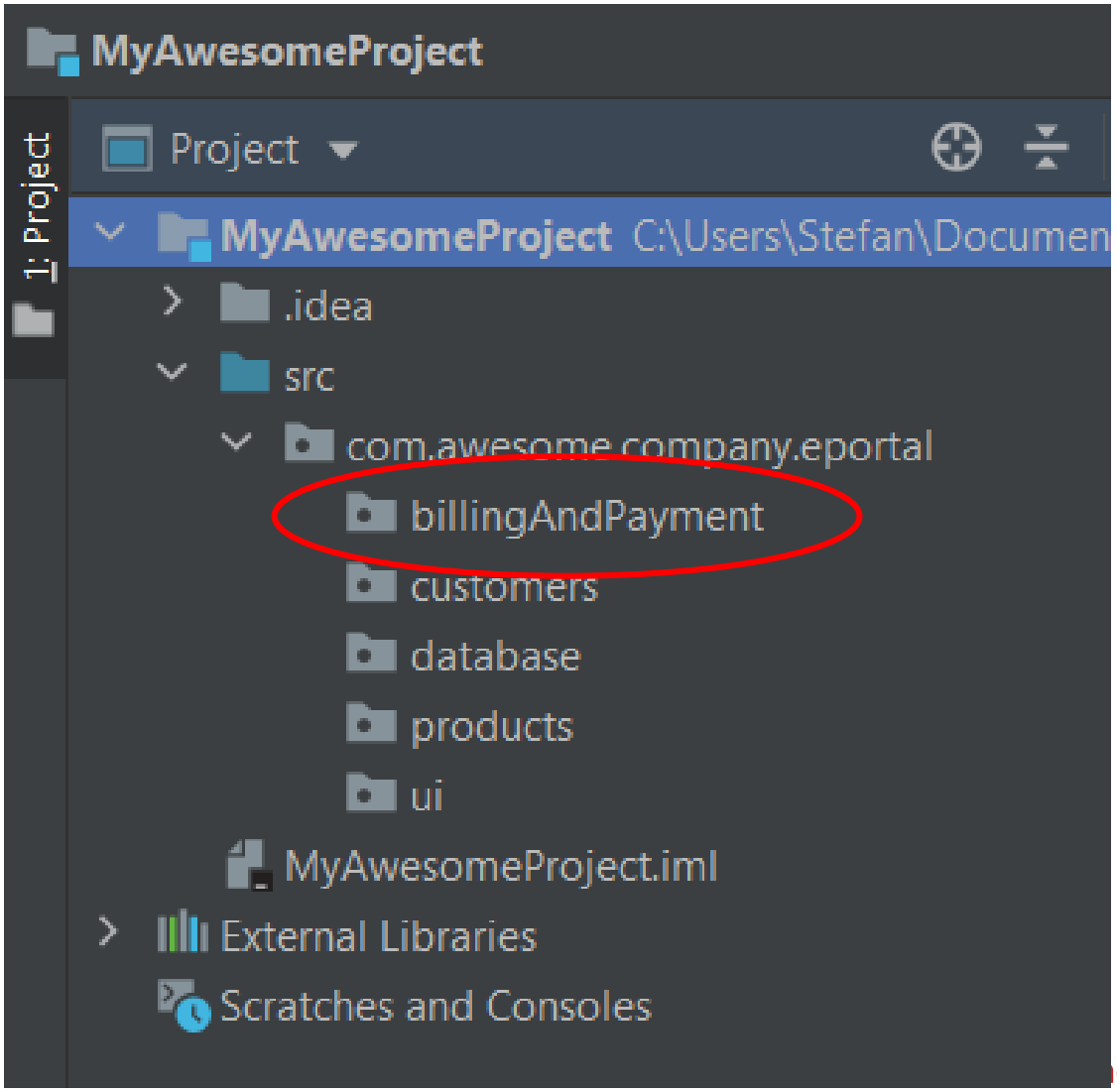




- Jedes Entity / Package / Modul / … ist für **genau einen Zweck** zuständig
- **+** weniger Wartung
- **+** besser verständlich

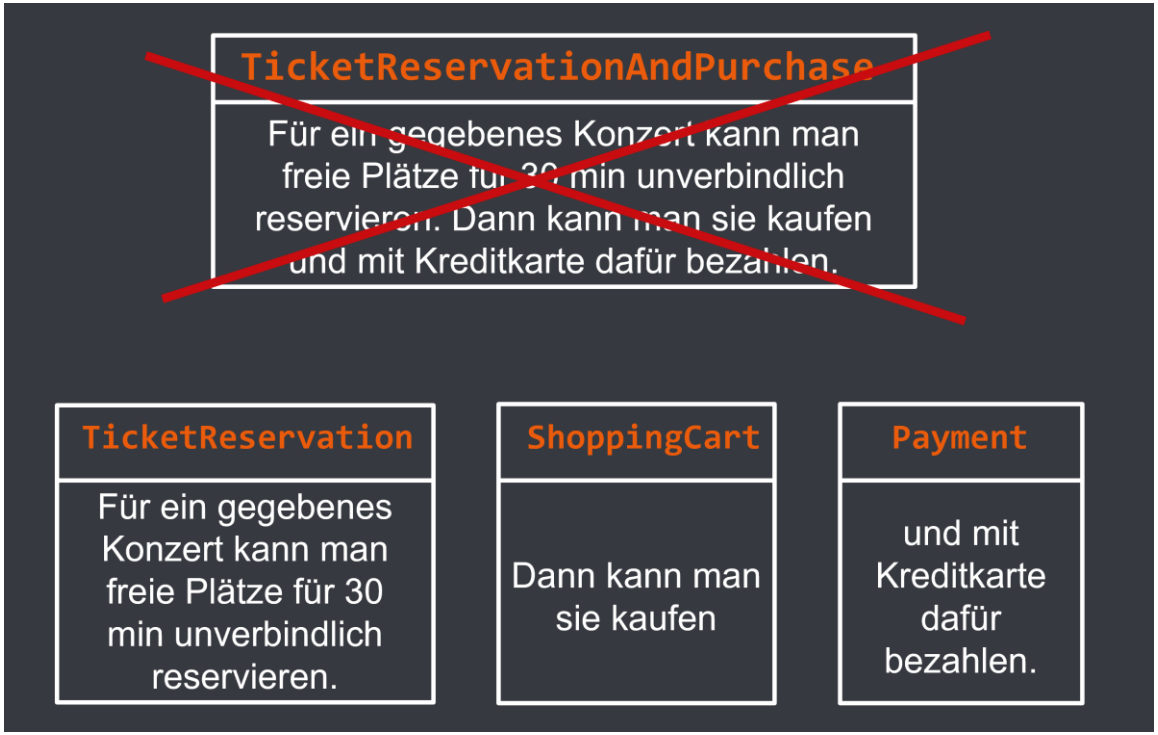
Beispiel für einen „Bad Smell“ bzgl. SRP

- Wenn im Namen eines Moduls, einer Klasse etc. ein „und“ oder „oder“ auftritt (bzw. „and“ / „or“), dann ist das ein starkes Indiz für zwei kombinierte Zwecke.
- Ein weiteres „Bad Smell“ Indiz sind „generische Sammel-Namen“ wie etwa
  - Common
  - Misc



SRP = Genau ein Grund für die Änderung einer Klasse

- SRP heißt auch: es gibt **genau einen Grund** für die Änderung einer Klasse (eines Moduls, Packages, Interfaces, …)
- Nehmen wir als Beispiel einen Webshop.
- Die obige Klasse „TicketReservationAndPurchase“ deckt den folgenden Use Case ab:
  - Für ein gegebenes Konzert kann man freie Plätze für 30 min unverbindlich reservieren. Dann kann man sie kaufen und mit Kreditkarte dafür bezahlen.
- Welche Gründe gäbe es für eine Änderung?
  1. Änderung der Reservierungszeit – 30 min sind doch zu lange; 10 min reichen.
  2. Neben Konzertkarten kann man auch Tshirts und CDs kaufen.
  3. Neben Kreditkarte soll eine Bezahlung auch mit Paypal möglich sein.

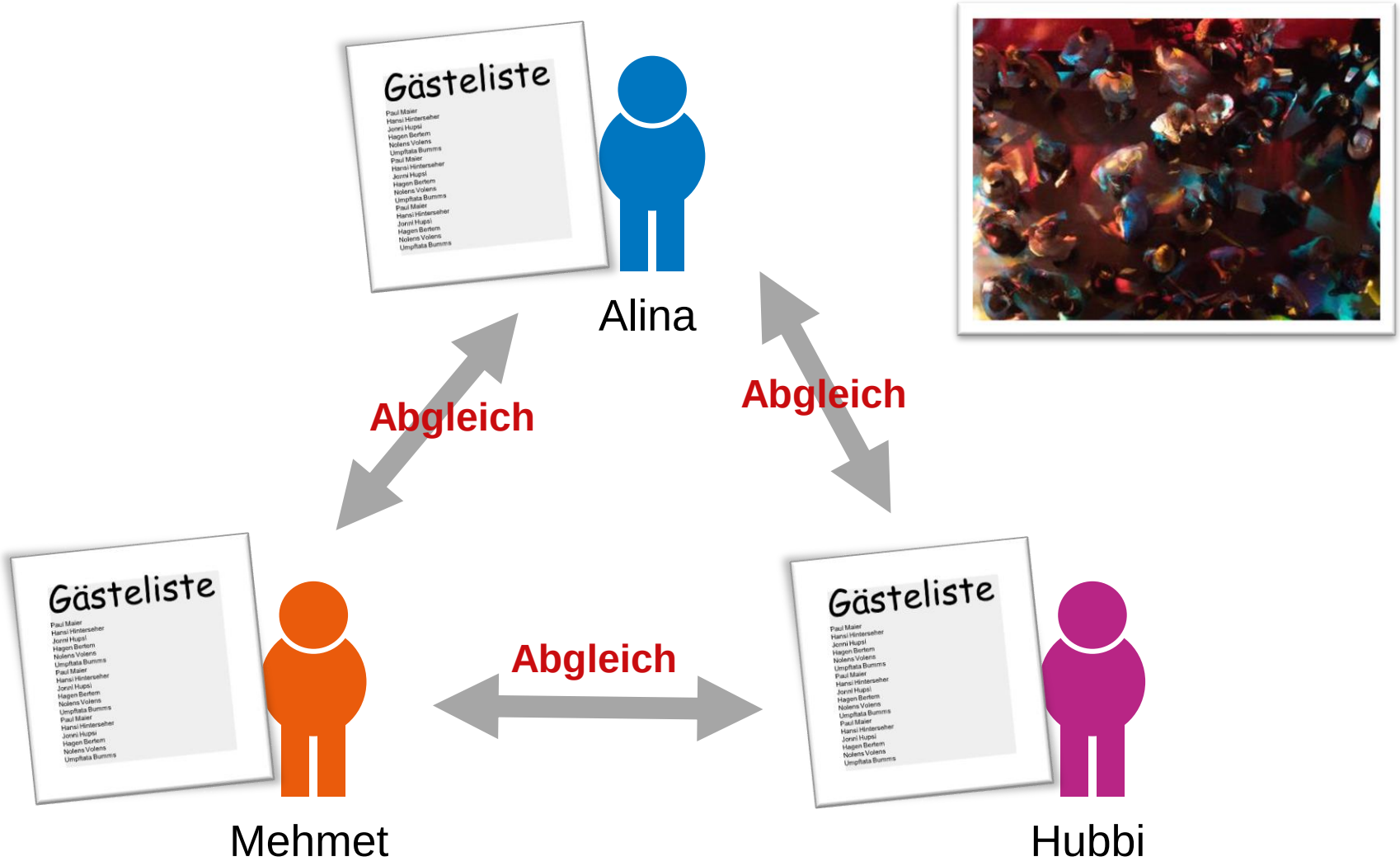


- Alle drei Änderungen treffen dieselbe Klasse – d.h. an dieser Klasse muss sehr oft etwas geändert werden. Das ist selten gut ☺.
- Wenn man die Klasse in die unteren drei Klassen auftrennt, dann gibt’s für jede dieser (kleineren) Klassen schon eher „nur einen Grund“ für eine Änderung.

2) Die nächste Party veranstalten Hubbi, Mehmet und Alina gemeinsam.

- Agneta sagt Mehmet erst ab
  - sagt dann aber gegenüber Alina doch noch zu
- Hubbi ist von Sabine frisch getrennt und lädt sie nicht ein
  - Alina weiß das noch nicht und lädt sie doch ein
- Mehmet lädt alle von der Fachschaft ein
  - …dadurch werden es 70 statt 50 Leute
  - Hubbi weiß das nicht und bucht einen Keller für 50

Eine gemeinsame Party – was ist hier passiert?



Besser: Nur eine führt die Gästeliste

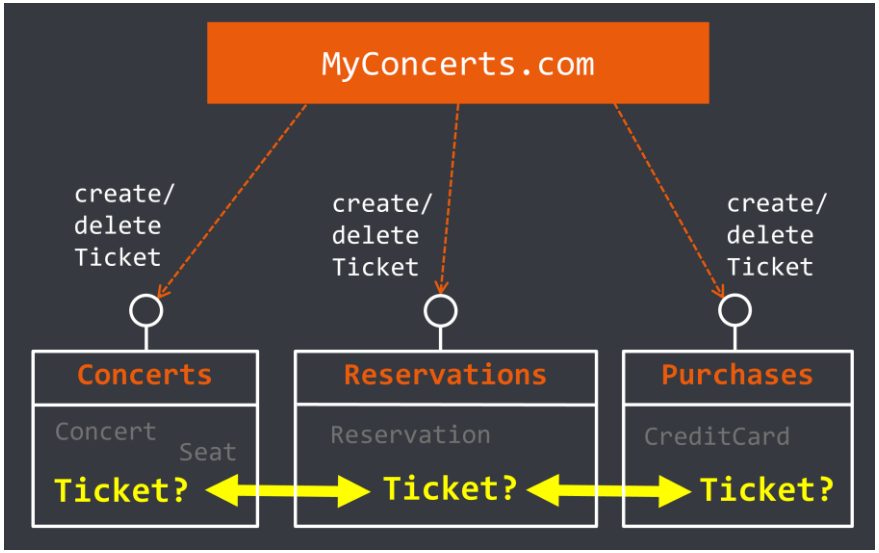
Alina

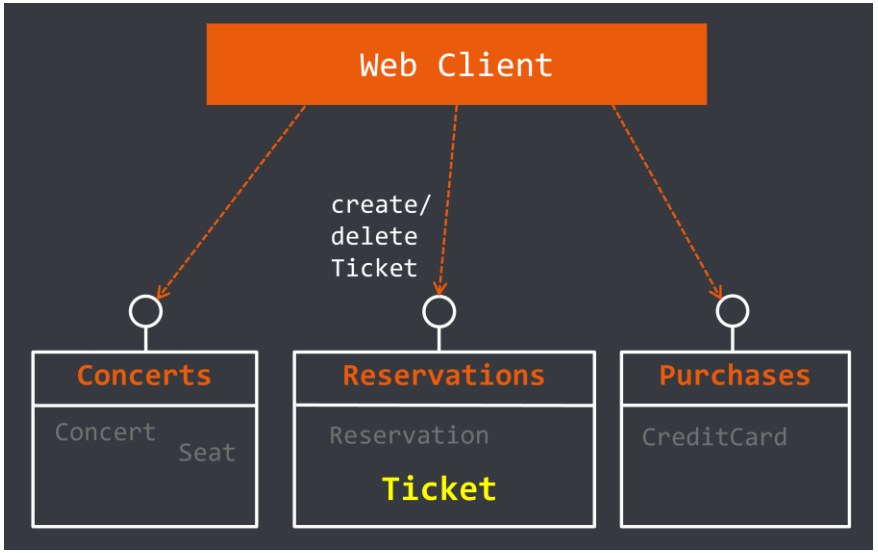
- Benachrichtigung "*Ich habe noch XYZ eingeladen*"
- Einsicht in Gästeliste

Mehmet

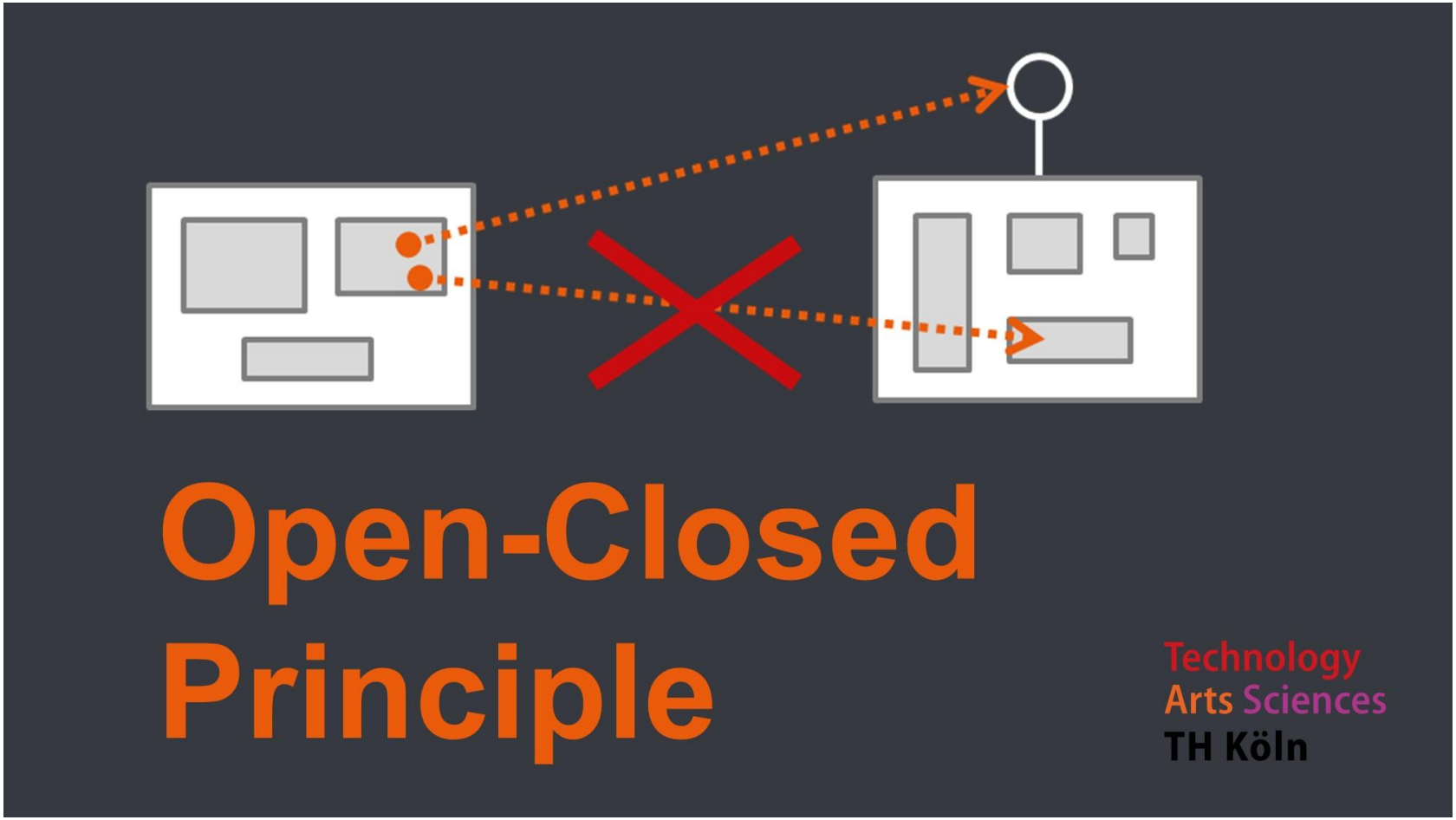
Hubbi

Entities “gehören” genau einem Modul





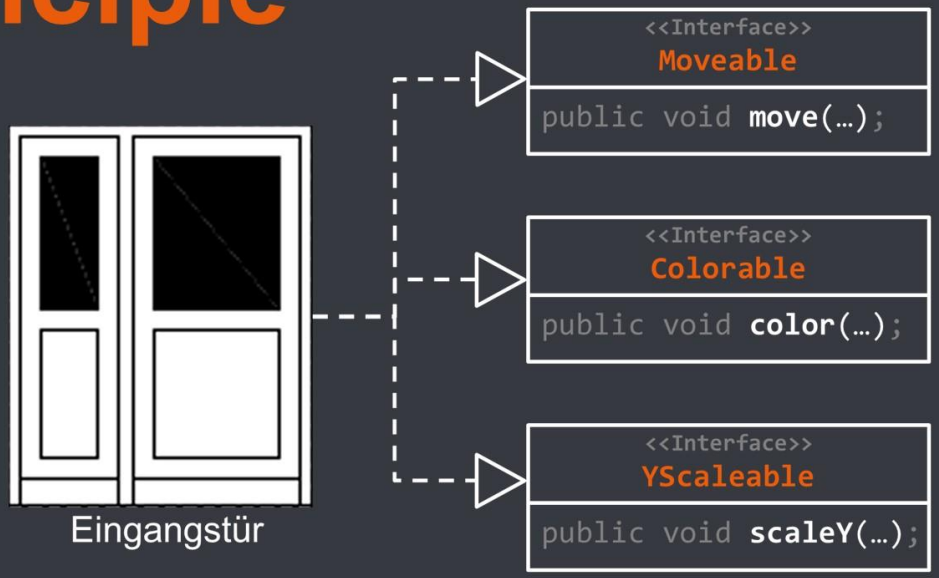
- gehören”: Create, Update, Delete nur hier
  - \+ weniger fehleranfällig
  - \+ besser verständlich
- Beispiel wieder unser Webshop. Nehmen wir an, es gibt Module “Concerts”, “Reservations” und “Purchases”.
- Wenn alle drei Module Tickets anlegen (create) oder ändern/löschen (update, delete) dürfen, dann müssen alle Änderungen immer sehr genau zwischen den Modulen synchronisiert werden, sonst gibt es Inkonsistenzen im Gesamtsystem (links).
  - Besser: nur ein Modul (Reservations) ist “Owner” von Ticket.
  - Nur dort werden Tickets angelegt oder gelöscht.
  - Das vermeidet die Synchronisation.



https://www.youtube.com/watch?v=udPiJpvPsMQ s

Das Script zu diesem Video folgt in Kürze …

Interface Segregation Principle



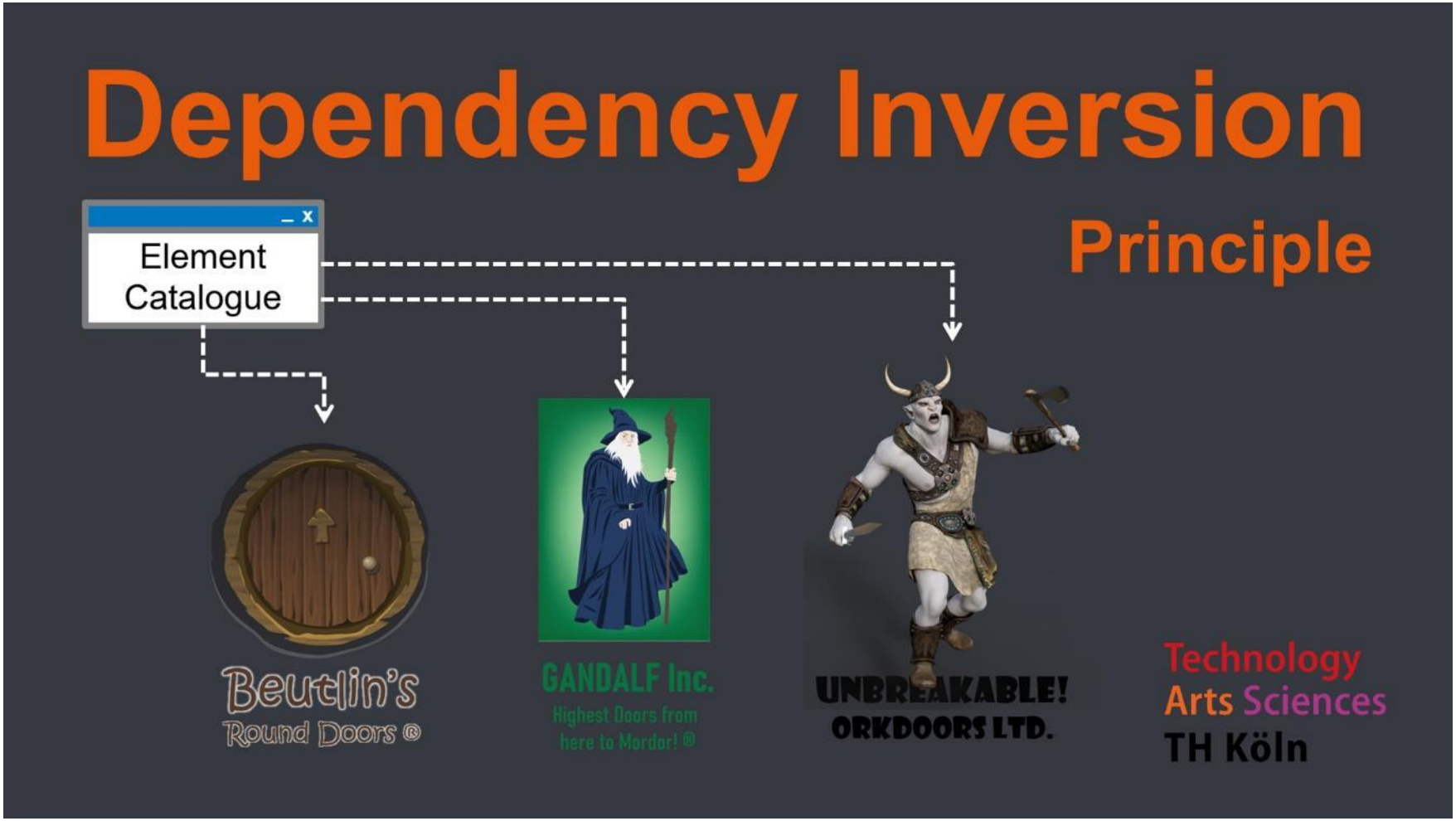
Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=HSSeLiqDXpg

Das Script zu diesem Video folgt in Kürze …

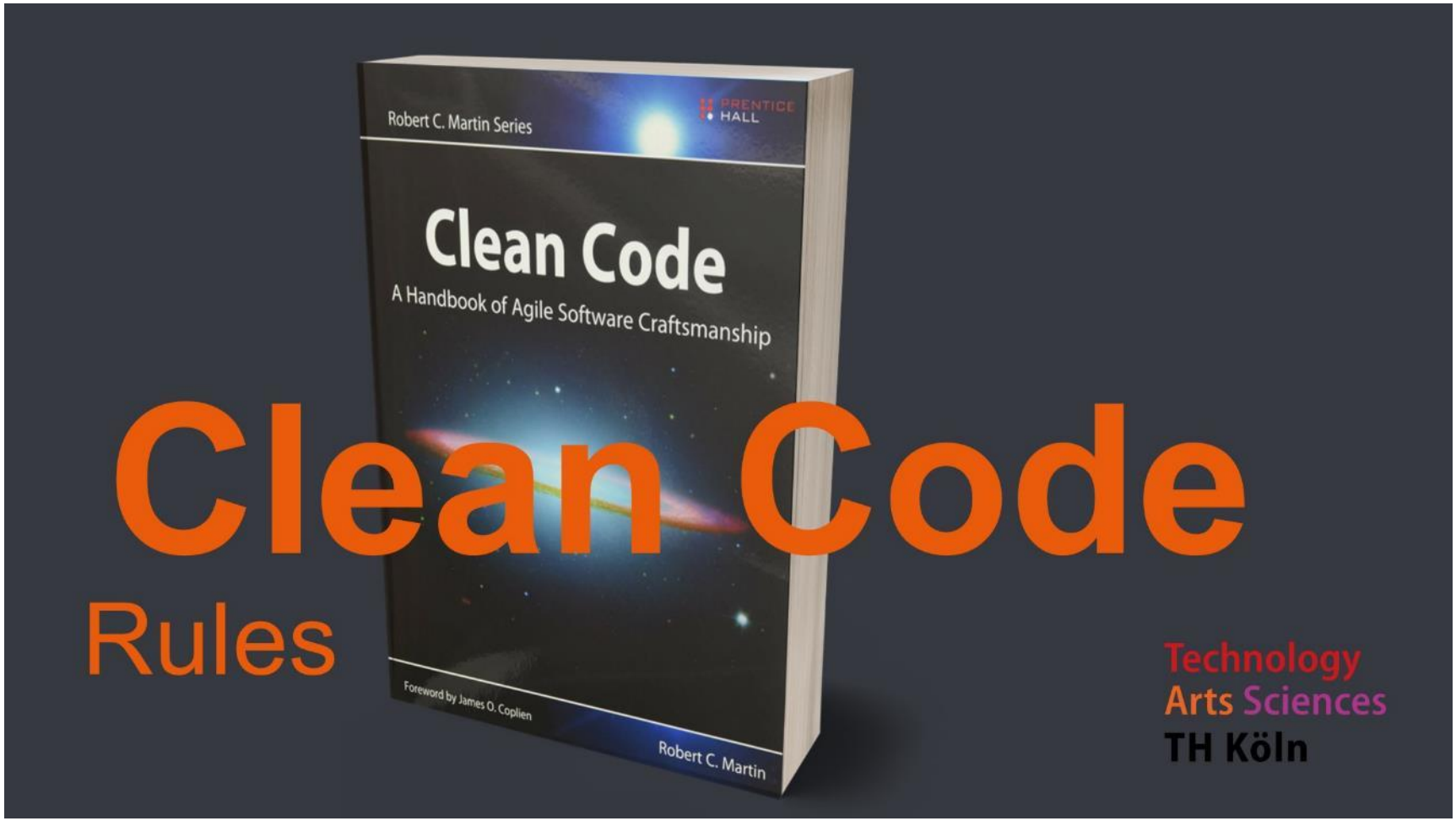
https://www.youtube.com/watch?v=Crdf_EDbDtA

Das Script zu diesem Video folgt in Kürze …



https://www.youtube.com/watch?v=swax9LubOec

Das Script zu diesem Video folgt in Kürze …



https://www.youtube.com/watch?v=2G20nqeHAn8

Das Script zu diesem Video folgt in Kürze …

1 SOLID Principles

1.1 Single Responsibility Principle

Criterium	Note
It should be possible to write the purpose of a class as one sentence on top of the class as a comment.	
Classes should have only one reason to change.	
The name of a class should describe what responsibilities it fulfills.	
Methods should do one thing, they should do it well, they should do it only.	
If a method does only "the next level of detail" compared to the method name, then the method is doing one thing	
Methods should not have sections inside them. If you are able to split a function into sections, then that method is probably doing multiple things.	

1.2 Open-Closed Principle, Interface Segregation Principle

Criterium	Note
Always use access control modifiers (public, protected, private) for classes, methods, and instance variables.	

Class organization should follow this order:
- Variables
  - Public static constants
  - private static variables
  - private instance variables
- Public method
  - Related private methods	

Classes should maintain Encapsulation. Variables and utility functions should be private.	
Objects should hide their data behind abstractions and expose functions that operate on that data.	

1.3 Dependency Inversion Principle

Criterium	Note
Classes should depend upon abstractions, not on concrete details.	
Hide implementation of classes with Abstraction. Have abstract interfaces that allow users to manipulate the data, without having to know its implementation.	

2 Clean Code Rules

2.1 Meaningful names

Criterium	Note
Name should tell you why it exists, what it does, and how it is used.	
Avoid using abbreviations (use `hypotenuse` instead of `hp`).	
Do not encode name with data structure (use `accounts` instead of `accountList`).	
Use constants instead of hard coding a value, `WORK_DAYS_PER_WEEK = 5` instead of just using 5.	
Classes and objects should have noun or noun phrase names. A class name should not be a verb.	
Methods should have verb or verb phrase names.	
The length of a name should correspond to the size of its scope. There can be a variable `i` inside a `for loop` but `i` should never be a `instance variable`.	
Don't add gratuitous context. For application "Gas Station Deluxe," it is a bad idea to prefix every class with `GSD`. For example use `AccountAddress` instead of `GSDAccountAddress`**.**	
Add no more context to a name than is necessary. Shorter names are generally better than longer ones, so long as they are clear.	

2.2 Comments only where necessary

Criterium	Note
Use a comment only if the code doesn't speak for itself.	
Always comment signatures in interfaces.	
Always comment on special conditions in the code if this is not obvious (e.g. why the order of statements matters).	

2.3 Keep your methods small

Criterium	Note
Methods should be small, not much longer than 20 (max 30) lines of code, and if possible should be much smaller.	
Lines should not be more than 80 (120 max) characters.	

Criterium	Note
The indent level of a function should not be greater than two or three.	

Rather than deeply nested if-statements, use guard clauses:
if (uploads.size() == 0) return DOCUMENTS_MISSING;
if (!uploadsChecked) return NOT_YET_CHECKED;
if (…) return …	

Use spaces between `operators`, `parameters`, and `commas`.	
In your methods, each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.	
Variables should be declared as close to their usage as possible.	

2.4 Only one level of abstraction within a method

Criterium	Note
We need to make sure that the statements within our method are all at the same level of abstraction.	

2.5 Stepdown rule

Criterium	Note
Try to order the methods in your class as if you tell a story. List them in the order they are called for the first time.	
The abstraction of a class should decrease as we go reading downwards.	

2.6 Proper error handling using exceptions

Criterium	Note
If your codebase has to many error handlers spread across different modules, then by default the code becomes unreadable.	
Use checked exceptions with care. If you throw a checked exception from a method in your code and the catch is three levels above, you must declare that exception in the signature of each method between you and the catch. Can be good sometimes, bad in other situations.	
Provide context with exceptions. Create informative error messages and pass them along with your exceptions. Mention the operation that failed and the type of failure.	

Criterium	Note
Define the exception in such a way that the caller can take a decision based on the exception only.	
Use different exception classes only if there are times when you want to catch one exception and allow the other one to pass through.	

2.7 General "Bad Smells" in the Code

Criterium	Note
No code duplication ("copy / paste"): means additional work, additional risk, and additional unnecessary complexity	
If you see commented-out code, delete it	
If you see unused code, delete it.	
Use Lombok where possible to avoid boiler-plate code.	
Base classes should know nothing about their derived classes.	
Code should be consistent: If within a particular method you use a variable named **response** to hold an HttpServletResponse, then use the same variable name consistently in the other methods that use HttpServletResponse objects.	
Prefer nonstatic methods to static methods.	
Avoid Negative Conditionals. `if (buffer.shouldCompact())` is better than `if (!buffer.shouldNotCompact())`	

Source: Derived from https://github.com/dev-aritra/clean-code-developer-checklist

https://www.youtube.com/watch?v=uiVXjRoa_Ys

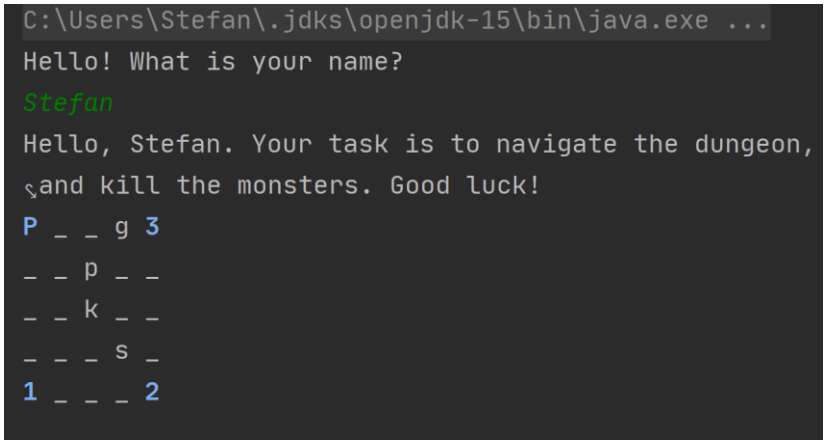
https://www.youtube.com/watch?v=GRMPFqRcq10

Aufgabenstellung: Ein Text Adventure

```
You wake up from a long, deep sleep and find  yourself on a beach.
> go south
You can't go that way, there is the ocean.
> go west
You are still on the beach.
> go west
You are still on the beach.
> go west
In the sand you see a box.
> open box
It's locked.
> take box
Taken.
> go north
You are about to enter a dense jungle. It smells funny.
> go north
A mighty ork appears in some distance. You smell his scent of rotten meat.
> go north
The ork catches you and eats you.
** GAME OVER **
```

- Ein Text Adventure ist ein einfaches Dungeon-Spiel, das nur auf der Console gespielt wird, in rein textlicher Form.

Die Aufgabenstellung im Detail



- Erlaubte Kommandos:
  - n — go North
  - e — go East
  - s — go South
  - w — go West
  - t <object> — take <object>
  - d <object> — drop <object>
  - u <object> — use <object>

- Es gibt zunächst nur **ein** Dungeon, das aber größer ist (5 x 5 oder größer).
- In diesem Dungeon befinden Sie sich (der Player) und mindestens ein Monster.
- Nach jedem Zug wird weiterhin das Dungeon ausgeprintet, aber es zeigt neben Ihrer Position auch das (oder die) Monster an.
- Auf jedem Feld des Dungeons kann immer nur ein "Lebewesen" stehen - egal ob Player oder Monster.
- Monster bewegen sich zufällig über das Dungeon, jeweils ein Feld in s/w/e/n Richtung. Sie können selbst entscheiden, ob Monster sich in jedem Zug bewegen oder seltener.
- Wenn der Player neben einem Monster steht, kann er
  - kämpfen, indem er eine Waffe benutzt ("u"),
  - oder etwas anders tun ("d" oder "t") - in diesem Fall wird das Monster ihn angreifen,
  - oder weglaufen.
- Waffen liegen als Items auf den Feldern des Dungeons, wo sie gefunden und aufgesammelt werden können.
  - Sie können sich selbst ausdenken, welche Waffen (in welcher Reihenfolge ...?) nötig sind, um das Monster zu besiegen.
- Meine Monster haben Zahlen (1 – 3). Außerdem werden für die Entwicklungsphase die Items mit ausgeprintet (das würde man natürlich später nicht mehr machen).

Einteilen der Entwicklungsschritte in Phasen

1. Erster Schritt: Die Entities
2. Erster Code (und ein bisschen Gold-Plating bei der Darstellung)
3. Modell: Nur noch ein einziger Items-Typ
4. Refactoring: Anwendung der Stepdown Rule
5. "Printable" interface
6. Nächster Entwicklungsschritt: Bewegen des Players
7. Refactoring: Verbesserung mit Command-Klasse und Exceptions
8. Refactoring: Zyklische Abhängigkeit zwischen Field und LivingCreature
9. Bewegen des Players - wie kommt man an das passende Field?
10. Refactoring: Blocker-Konzept bei Feldern konsequenter umsetzen
11. Refactoring: Exceptions
12. Nächstes Feature: Items vom Boden aufheben
13. Nächstes Feature: Items fallen lassen
14. Nächstes Feature: Monster greifen an
15. Refactoring: Stepdown Rule / Keep Methods Small in Game.play()
16. Nächstes Feature: Player greift Monster an
17. Letztes Feature: Monster bewegen sich
18. Refactoring: Package-Struktur
19. Revisiting the Model

- Betrachtet man die Entwicklungsschritte, dann ergibt sich folgendes Muster:
  - **Blau**: Modellierung – also im Wesentlichen die Arbeit mit einem Stück Papier. Am Anfang, zur Konzeption, und dann nochmal am Ende, um eine kurze visuelle Dokumentation als Diskussionsgrundlage mit Kollegen zu haben.
  - **Orange:** Implementieren von Features
  - **Grün:** Refactoring – also Verbessern von Code, der einem nicht gefällt – Abtragen von „technischer Schuld“, wenn man so will.
- Ohne dass ich es vorab geplant hatte, wechseln sich **orange** (Entwicklung) und **grün** (Refactoring / Abtragen technische Schuld) miteinander ab.

… siehe „Sägezahn-Ansatz” von Carola Lilienthal

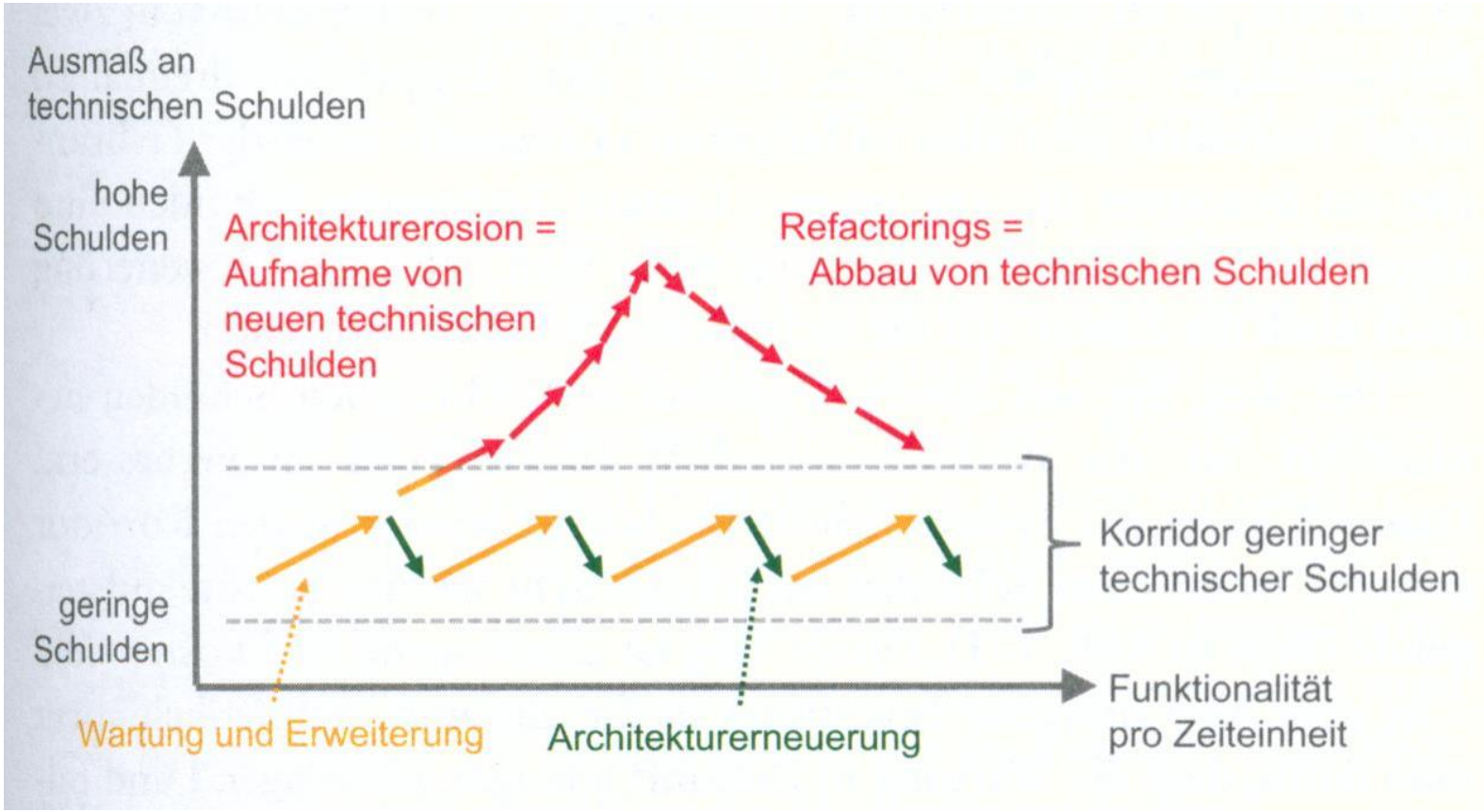


Bild: [Lilienthal], S. vi

- Das entspricht ganz dem “Sägezahn-Ansatz” von Carola Lilienthal – siehe Video zu “Prinzipien, Patterns, Architekturstile”.

Eine Beispiellösung für das Text-Adventure

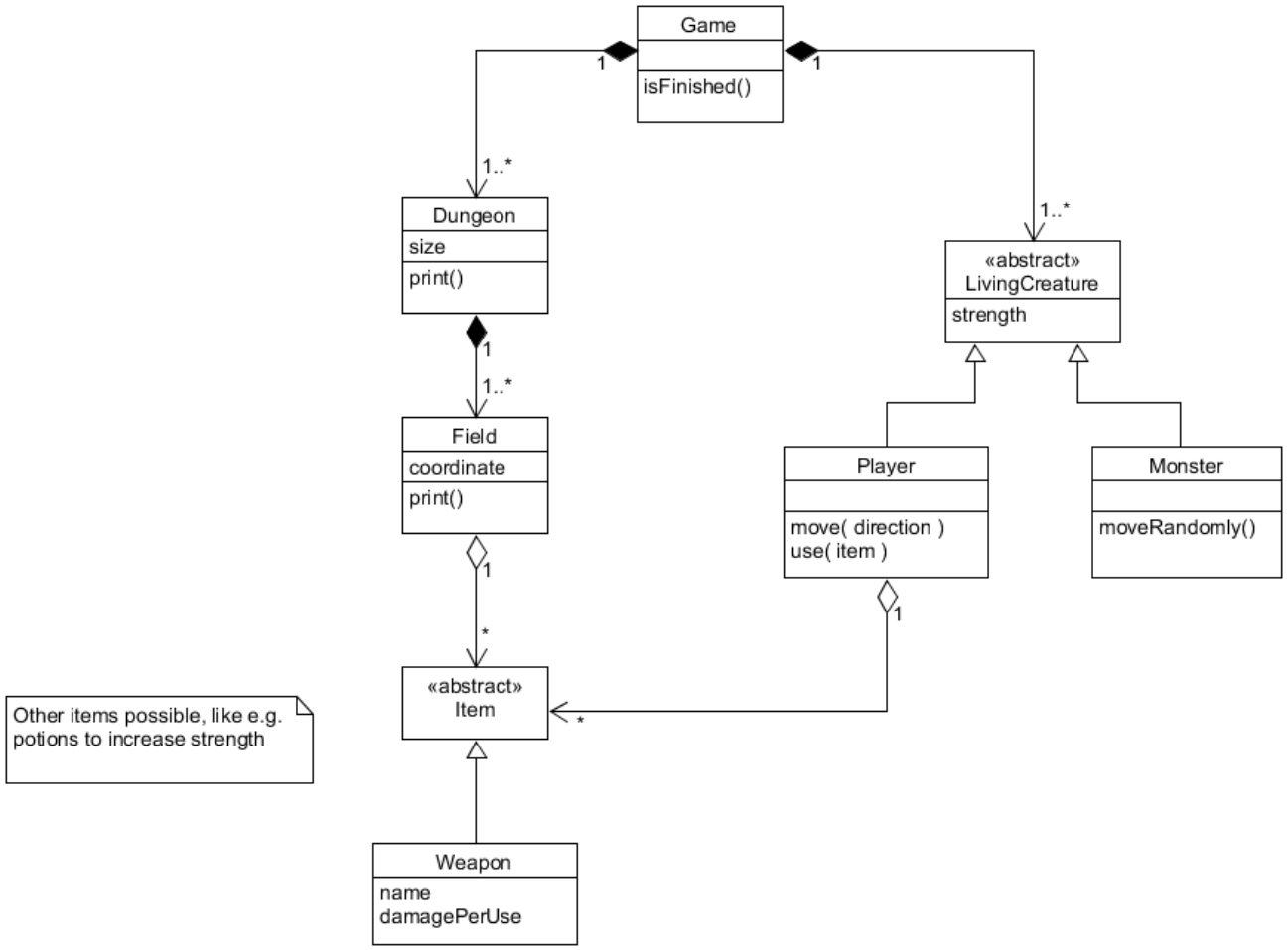
Eine Beispiellösung für das Text-Adventure

Nachfolgend stelle ich Ihnen dar, wie eine Beispiellösung für das Text-Adventure aussehen kann. Die Beispiellösung gibt's hier im Repo: https://git.st.archi-lab.io/students/st2/ss21/exercises/textadventure-lecture-example

Ich habe dabei mal bewusst versucht, meinen "Denkprozess" abzubilden. Das mache ich nicht, weil ich den so einzigartig finde 😅. Da gibt es in jedem Fall auch andere sinnvolle Herangehensweisen. Aber vielleicht hilft es Ihnen, wenn ich versuche, meine Denkprozesse transparent zu machen.

1. Erster Schritt: Die Entities

Als erstes habe ich mir überlegt, welche Entities ich gern in meinem Spiel hätte. Das fand ich ziemlich leicht und "straightforward". Hier die erste Version des Modells:



Ein paar Prinzipien fand ich aus der Aufgabenstellung heraus recht klar:

- Player und Monster haben Gemeinsamkeiten, brauchen also eine gemeinsame Abstraktion
  - belegen ein Feld exklusiv
  - haben eine "Strength", die durch Kämpfe u.a. verändert wird.
  - Letzteres war für mich das Argument, da eine abstrakte Klasse draus zu machen.
- Game enthält den Game-Loop und die Initialisierung
- Dungeon ist ein Entity, und Field auch, damit ein Field Dinge (Items) enthalten kann

Anderes habe ich erstmal offen gelassen:

1. Wo und wie passiert die Kommandoverarbeitung?

2. Wie genau ist die Datenstruktur im Dungeon für die Fields? Array?
3. Wie wird ein Kampf abgebildet (und dass verschiedene Waffen verschiedene Effekte haben?)

2. Erster Code (und ein bisschen Gold-Plating bei der Darstellung)

Als erstes habe ich dann die Entities angelegt und angefangen, die Methoden und Abstraktionen umzusetzen. Dabei war "Printable" ein guter Startpunkt, weil man das sowieso braucht und es einen guten Rahmen für das Game schafft. Sprich: Ich hatte einen gut definierten "Meilenstein", der aus statisch ausgeprinteten Spielfeld bestand.

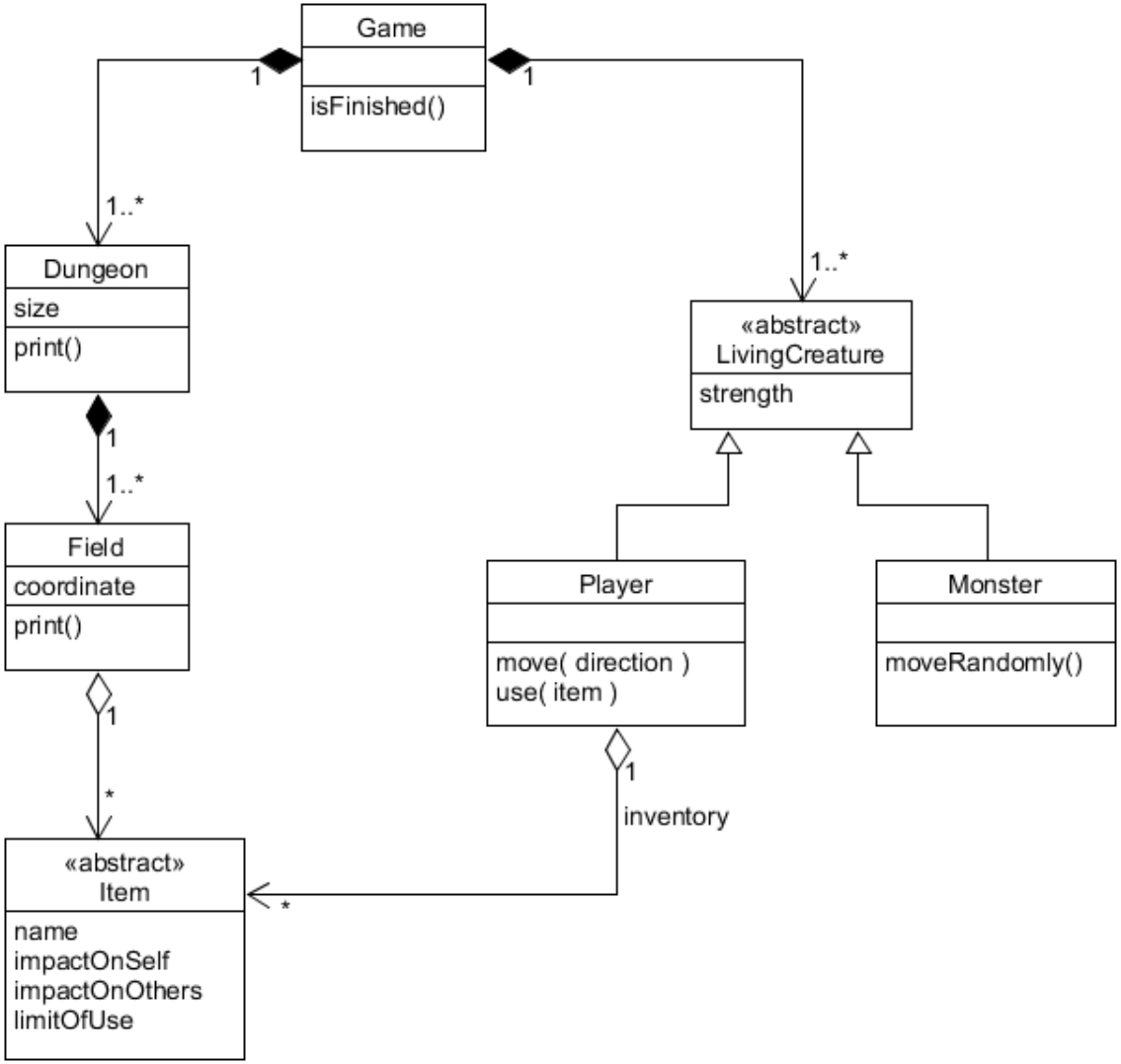
Etwas Gold-Plating konnte ich mir nicht verkneifen: Player und Monster werden in Farbe ausgegeben, und die Farbe ändert sich mit der relativen Strength. Damit musste ich dann auch direkt das Strength-Modell machen, was auch ganz hilfreich war.

Die Frage nach der Datenstruktur Dungeon - Field musste dann wegen dem Printen schnell entschieden werden - fürs Erste als Array. Wenn das nicht trägt, wird es geändert. Dann noch die Init-Methode in Game mit dem Player auf 0,0 und Monstern in allen vier Ecken - ging glatt durch und ohne erkennbare "bad smells" im Code.

```
C:\Users\Stefan\.jdks\openjdk-15\bin\java
P o o o M
o o o o o
o o o o o
o o o o o
M o o o M

Process finished with exit code 0
```

3. Modell: Nur noch ein einziger Items-Typ

Bei den Items schien es so, dass man das Modell ein bisschen einfacher machen konnte - mit "impactOnSelf" und "impactOnOthers" sowie "limitOfUse" waren sowohl Potions wie auch Grenades abgedeckt. Sogar Sprengstoffgürtel, wenn man wollte 👷.



4. Refactoring: Anwendung der Stepdown Rule

Beim Platzieren der Items dann die erste Anwendung der Stepdown Rule / Keep Methods Small. Game.initialize() war jetzt schon ziemlich lang:

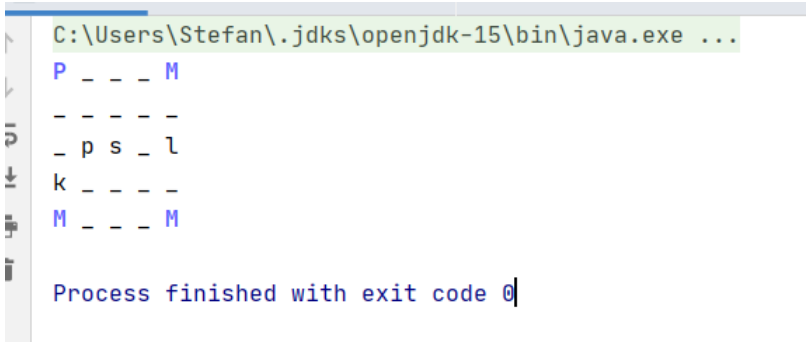
```
public void initialize() {
    dungeon = new Dungeon( 5, 5);
    player = new Player( 50.0f );
    dungeon.getField(0, 0 ).setCreature( player );

    // monsters in all the other corners:
    Monster monster = new Monster( 50.0f );
    monsters.add( monster );
    dungeon.getField(0, 4 ).setCreature( monster );
    monster = new Monster( 100.0f );
    monsters.add( monster );
    dungeon.getField(4, 4 ).setCreature( monster );
    monster = new Monster( 150.0f );
    monsters.add( monster );
    dungeon.getField(4, 0 ).setCreature( monster );

    // some weapons and potions
    dungeon.getField(2, 1 ).setItem(
       new Item( "potion", +10.0f, 0f, 1 )
    );
    dungeon.getField(3, 0 ).setItem(
            new Item( "knife", 0.0f, -5.0f )
    );
    dungeon.getField(2, 2 ).setItem(
            new Item( "sword", 0.0f, -20.0f )
    );
    dungeon.getField(2, 4 ).setItem(
            new Item( "lasergun", 0.0f, -60.0f )
    );
}
```

Also Refactoring und das Platzieren von Monstern und Items rausgezogen:

```
public void initialize() {
    dungeon = new Dungeon( 5, 5);
    player = new Player( 50.0f );
    dungeon.getField(0, 0 ).setCreature( player );
    placeMonsters();
    placeItems();
}

private void placeMonsters() {
    // monsters in all the other corners:
    Monster monster = new Monster( 50.0f );
    monsters.add( monster );
    dungeon.getField(0, 4 ).setCreature( monster );
    monster = new Monster( 100.0f );
    monsters.add( monster );
    dungeon.getField(4, 4 ).setCreature( monster );
    monster = new Monster( 150.0f );
    monsters.add( monster );
    dungeon.getField(4, 0 ).setCreature( monster );
}

private void placeItems() {
    dungeon.getField(2, 1 ).setItem(
            new Item( "potion", +10.0f, 0f, 1 )
    );
    dungeon.getField(3, 0 ).setItem(
            new Item( "knife", 0.0f, -5.0f )
    );
    dungeon.getField(2, 2 ).setItem(
            new Item( "sword", 0.0f, -20.0f )
    );
    dungeon.getField(2, 4 ).setItem(
            new Item( "lasergun", 0.0f, -60.0f )
    );
}
```

5. "Printable" interface

Das Printable Interface bewährt sich jetzt schon als Abstraktion, weil man einfach das Drucken wunderbar hierarchisch durchreichen kann. Um während der Entwicklung zu sehen, wo meine Items liegen, drucke ich sie mit aus. Das ist das Ergebnis:



Dafür brauche ich folgenden Code:

Printable

```
package thkoeln.archilab.exercises.textadventure;

public interface Printable {
    public void print();
}
```

Game.main(...)

```
game.getDungeon().print();
```

Dungeon

```
@Override
public void print() {
    for ( int x = 0; x < width; x++ ) {
        for ( int y = 0; y < height; y++ ) {
            fields[x][y].print();
            System.out.print( " " );
        }
        System.out.println( "" );
    }
}
```

Field

```
@Override
public void print() {
    if( creature != null ) {
        creature.print();
    }
    else if( item != null ) {
        item.print();
    }
    else {
        System.out.print( "_" );
    }
}
```

LivingCreature

```
public abstract class LivingCreature implements Printable {
    //...
    public abstract String printSymbol();

    @Override
    public void print() {
        ColorGradientPrinter.colorPrint( printSymbol(), getRelativeStrength() );
    }
}
```

Unschön: Die Formatierung des Dungeons ist über mehrere Klassen verstreut; das könnte man nochmal angehen. Später.

6. Nächster Entwicklungsschritt: Bewegen des Players

Als nächsten "Teilmeilenstein" möchte ich den Player im Dungeon bewegen können. Dafür muss ich mir überlegen, wo der Command Processor hinkommt. Naheliegendste Lösung:

- in Game
- Dort wird einem Loop wird so lange ein Kommando eingelesen, bis die Abbruchbedingung erfüllt ist

Als erste Implementierung des Command Processors habe ich erstmal eine einfache Lösung gewählt:

```
public void play() {
    while( !isFinished() ) {
        String[] command = nextCommand();
        String outMsg = "You wrote: '" + command[0] + "'";
        if (command.length > 1) {
            outMsg += "' and then the word '" + command[1] + "'";
        }
        System.out.println(outMsg);
    }
}

public boolean isFinished() {
    return false;
}


private String[] nextCommand() {
    while( true ) {
        String line = scanner.nextLine();
        String[] parts = line.split("[ \t]+");
        if ((parts.length < 1 || parts.length > 2) || (parts[0].length() != 1)) {
            System.out.println("*** Invalid input! Please try again. ***");
        }
        return parts;
    }
}
```

Allerdings hat mich das nicht wirklich zufrieden gemacht:

- String[] ist keine gute Datenstruktur, besser wäre eine echte Command-Klasse
- ... und statt der Outputs wären Exceptions besser.

7. Refactoring: Verbesserung mit Command-Klasse und Exceptions

Wir führen einen Enum für die verschiedenen Kommando-Typen ein.

CommandType

```
public enum CommandType {
    s, e, w, n, u, t, d;

    public boolean isMoveCommand() {
        return ( this == s || this == n || this == e || this == w );
    }
}
```

Game

So sieht das in der Game-Klasse schon besser aus:

```
private Command nextCommand() {
    Command command = null;
    while( command == null ) {
        String line = scanner.nextLine();
        try {
            command = Command.valueOf(line);
        } catch (CommandException exception) {
            System.out.println(exception.getMessage() + " Please try again.");
        }
    }
    return command;
}
```

Die ganze Verarbeitung des String-Inputs mit angemessener Fehlerprüfung findet dann in einer Command-Factory-Methode statt (Command.valueOf(...)):

Command

```
public static Command valueOf(String inputFromCommandLine ) {
    if( inputFromCommandLine == null ) throw new CommandException( "Invalid command line!" );
    String[] parts = inputFromCommandLine.split("[ \t]+");
    if ( parts.length < 1 ) throw new CommandException( "Empty input!" );
    if ( parts.length > 2 ) throw new CommandException( "Max. 2 words allowed!" );
    if ( parts[0].length() != 1 ) throw new CommandException( "First word must be 1-letter command!" );

    CommandType commandType;
    try {
        commandType = CommandType.valueOf( parts[0] );
    }
    catch ( IllegalArgumentException exception ) {
        throw new CommandException( "Unknown command!" );
    }
    if ( commandType.isMoveCommand() ) {
        return new Command( commandType, null );
    }
    else {
        if (parts.length < 2) throw new CommandException("Item name needs to be specified for a non-move command!");
        return new Command(commandType, parts[1]);
    }
}
```

Die Methode könnte etwas kürzer sein, aber mir fällt kein guter inhaltlicher Split ein - also bleibt das erstmal so. Dafür deckt sie auch tatsächlich alle Fehlerfälle ab. Das einzige, was noch nicht geprüft wird, ist ob der "itemName" auch existiert (d.h. gibt es auf dem Feld tatsächlich das Item "itemName", das man gerade aufheben will). Die Prüfung passt hier auch nicht so gut hin. Die machen wir da, wo wir ein Feld oder das Inventory kennen.

8. Refactoring: Zyklische Abhängigkeit zwischen Field und LivingCreature

Jetzt ist alles zusammen, um Kommandos auch ausführen zu können. Dazu kann man in Player eine "execute(Command command)" Methode anlegen, die dann wiederum dedizierte Methoden aufruft.

Player

```
public void execute( Command command ) {
    if ( command.getCommandType().isMoveCommand() ) {
        move( command.getCommandType() );
    }
    if ( command.getCommandType() == CommandType.t ) take( command.getItemName() );
    if ( command.getCommandType() == CommandType.d ) drop( command.getItemName() );
    if ( command.getCommandType() == CommandType.u ) use( command.getItemName() );
}
```

Allerdings stellt sich die Frage nach der zyklischen Abhängigkeit zwischen Field und LivingCreature. Zwar kann man LivingCreature schon als Abstraktion sehen, aber besser scheint es mir, hier noch ein Interface einzuziehen, das bei Field die Abfrage nach "blockiertem Feld" regeln kann. Das kann man mit einem "Blocking" Interface machen.

Blocking Interface

```
public interface Blocking extends Printable {
    public void place( Field field );
}
```

Field

```
public void setInhabitant( Blocking inhabitant ) {
    if( this.inhabitant != null ) {
        throw new AlreadyBlockedException();
    }
    this.inhabitant = inhabitant;
}
```

Platzieren von Player oder Monstern

Das Platzieren von Player oder Monstern sollte dann auch verschoben werden. Statt wie jetzt dem Field zu sagen, dass eine LivingCreature draufsteht ....

```
private void placeMonsters() {
    // monsters in all the other corners:
    Monster monster = new Monster( 50.0f );
    monsters.add( monster );
    dungeon.getField(0, 4 ).setLivingCreature( monster );
    monster = new Monster( 100.0f );
    monsters.add( monster );
    dungeon.getField(4, 4 ).setLivingCreature( monster );
    monster = new Monster( 150.0f );
    monsters.add( monster );
    dungeon.getField(4, 0 ).setLivingCreature( monster );
}
```

... sollte man besser die LivingCreature auf ein Field setzen:

```
private void placeMonsters() {
    // monsters in all the other corners:
    Monster monster = new Monster( 50.0f  );
    monster.place( dungeon.getField(0, 4 ) );
    monsters.add( monster );

    monster = new Monster( 100.0f );
    monster.place( dungeon.getField(4, 4 ) );
    monsters.add( monster );

    monster = new Monster( 150.0f );
    monster.place( dungeon.getField(4, 0 ) );
    monsters.add( monster );
}
```

9. Bewegen des Players - wie kommt man an das passende Field?

Die move-Methode ist im Player schon (leer) angelegt, aber es gibt noch ein Problem. Wie komme ich ausgehend vom "Field" an das entsprechende Nachbarfeld? Die Fields hängen einfach als Array im Dungeon. Idee: ich lege eine Hashmap in Field an, die die Nachbarn enthält (neighbours).

Field

```
HashMap<CommandType, Field> neighbours = new HashMap<>();
```

Nicht 100% elegant, weil ich die Himmelsrichtungen und die CommandTypes (von denen es ja noch ein paar mehr gibt) hier zusammenwerfe. Leider kann man aber enums nicht von einander ableiten, sonst würde ich Directions (n,s,e,w) definieren und diese dann zu CommandType erweitern. Wie auch immer - das Problem löse ich später.

Die neighbours werden im Dungeon-Konstruktur initialisiert, wo die Fields instanziiert werden. Auch nicht 100% elegant, weil ich gern eine besser gekapselte Lösung hätte. Aber so geht es erstmal.

Dungeon

```
Dungeon( int width, int height ) {
    this.width = width;
    this.height = height;
    fields = new Field[width][height];
    for ( int x = 0; x < width; x++ ) {
        for ( int y = 0; y < height; y++ ) {
            fields[x][y] = new Field();
            // as we move from left to right and from top to bottom, this method of setting
            // neighbours is sufficient - provided all neighbours are properly initialized to null.
            if( x > 0 ) fields[x][y].setWestNeighbour( fields[x-1][y] );
            if( y > 0 ) fields[x][y].setNorthNeighbour( fields[x][y-1] );
        }
    }
}
```

Die *setWestNeighbour* und *setNorthNeighbour* Methoden setzen die Nachbarschaftsbeziehungen jeweils gegenseitig.

10. Refactoring: Blocker-Konzept bei Feldern konsequenter umsetzen

LivingCreature hat jetzt eine "protected" move-Methode (protected, damit die jeweilige Instanz selbst entscheiden kann, welche Arten von Moves "ok" sind. Bei Player z.B. nur aufgrund eines Kommandos. Man kann also move nicht von außen aufrufen.)

LivingCreature

```
// Only protected, not public, as it should not be set by the outside world.
// Instead, outside calls should use some dedicated move command. Only instances of
// LivingCreature may directly call this method.
protected void move(CommandType direction ) {
    Field newField = currentField.getNeighbours().get( direction );
    if ( newField == null ) {
        throw new CommandException( "You can't go this way!" );
    }
    else {
        newField.setInhabitant( this);
        currentField.setInhabitant( null );
        currentField = newField;
    }
}
```

Hier fällt auf, dass das Konzept so nicht so schön umgesetzt ist.

1. Es ist unschön, setInhabitant(null) aufzurufen - besser wäre eine dedizierte Methode, um ein Feld leer zu machen.
2. Das gleiche Konzept hat 2 Namen - Blocking und Inhabitant (Verstoß gegen konzeptuelle Integrität, verwandt mit Liskov Substitution Principle).

Der Deutlichkeit halber mache ich ein Interface daraus und benenne die Methoden entsprechend um:

Blockable

```
public interface Blockable {
    /**
     * Blocker will be removed.
     */
    void unblock();

    /**
     * New blocker added
     * @param blocker
     */
    void block(Blocking blocker);
}
```

Field

```
public class Field implements Printable, Blockable {
    //...
    @Override
    public void unblock() {
        this.blocker = null;
    }

    @Override
    public void block(Blocking blocker) throws AlreadyBlockedException {
        if( this.blocker != null ) {
            throw new AlreadyBlockedException( "This field is already blocked!" );
        }
        this.blocker = blocker;
    }
    //...
}
```

11. Refactoring: Exceptions

Bis jetzt waren alle Exceptions RuntimeExceptions, die nicht deklariert werden müssen. Das ist einerseits einfacher, andererseits auch weniger explizit. Ich mache mal den Versuch, alles auf "checked exceptions" umzustellen - falls mein Code durch viele try/catch-Blöcke verschandelt wird, dann lasse ich das wieder. Die Hoffnung ist, dass ich das nur einmal - oben im Game-Loop - fangen muss. Das funktioniert auch ganz gut:

Command

```
public class Command {
    public static Command valueOf(String inputFromCommandLine ) throws CommandException {
        //...
    }
}
```

Game

nextCommand reicht die Exception noch weiter nach oben ...

```
private Command nextCommand() throws CommandException {
    Command command = null;
    while( command == null ) {
        String line = scanner.nextLine();
        command = Command.valueOf(line);
    }
    return command;
}
```

... und im Game-Loop (Methode "play()") wird sie dann gefangen und behandelt (indem ausgegeben wird: "<Error Message>. Please try again.").

```
public void play() {
    while( !isFinished() ) {
        dungeon.print();
        System.out.println();
        try {
            Command command = nextCommand();
            player.execute(command);
            System.out.println();
        }
        catch ( CommandException commandException ) {
            System.out.println( commandException.getMessage() );
            System.out.println( "Please try again.\n");
        }
    }
}
```

12. Nächstes Feature: Items vom Boden aufheben

Es sollte eigentlich alles vorbereitet sein, um auch Items zu erkennen und aufzuheben. Das ist auch so - es reichen 6 Zeilen in der Player.take(...) Methode.

Player

Es gibt eine "execute" Methode, die beliebige Kommandos auswerten und an entsprechende move(...), take(...), drop(...) oder use(...) Methoden delegiert.

```
public void execute( Command command ) throws CommandException {
    if ( command.getCommandType().isMoveCommand() ) {
        move( command.getCommandType() );
    }
    if ( command.getCommandType() == CommandType.t ) take( command.getItemName() );
    if ( command.getCommandType() == CommandType.d ) drop( command.getItemName() );
    if ( command.getCommandType() == CommandType.u ) use( command.getItemName() );
}
```

Die take(...)-Methode funktioniert wie man es sich vorstellt: Eine CommandException, wenn es das genannte Item hier nicht gibt. Ansonsten einfach Item vom Field nehmen und dem Inventory zufügen.

```
private void take( String itemName ) throws CommandException {
    if ( !getCurrentField().containsItem() ||
         !getCurrentField().getItem().getName().equals( itemName) )
              throw new CommandException( "Here is no " + itemName + "." );
    inventory.put( itemName, getCurrentField().getItem() );
    getCurrentField().setItem( null );
    System.out.println( "You have picked up a " + itemName + "." );
}
```

Das einzige, was etwas stört: Wenn man versucht, etwas aufzuheben, was **nicht** da ist, wird die Meldung (über die Exception) im Game-Loop ausgegeben. Die Erfolgsmeldung erfolgt lokal. Das sollte man auch nochmal ändern, damit man die Kommunikation über eine Methode bündeln kann. Für den Moment lasse ich das so.

13. Nächstes Feature: Items fallen lassen

Beim Fallenlassen muss man bedenken, dass Felder momentan nur ein Item können. Geht aber mit der bisherigen Struktur prima:

Player

```
private void drop( String itemName ) throws CommandException {
    if ( !inventory.containsKey( itemName ) )
        throw new CommandException( "You don't have a " + itemName + "." );
    if ( getCurrentField().containsItem() )
        throw new CommandException( "There is no room to drop it." );
    getCurrentField().setItem( inventory.get( itemName ) );
    inventory.remove( itemName );
}
```

14. Nächstes Feature: Monster greifen an

Auch das geht jetzt recht leicht. Als zusätzliche Abstraktion definiere ich ein Interface *Impactable*, das ausdrückt, das man von außen angegriffen oder geheilt werden kann.

Impactable

```java
public interface Impactable {

    /**
     * Receive an external impact (positive by healing, or negative by attack)
     * @param impact if positive, it is a healing, otherwise it is an attack
     */
    public void receiveImpact( float impact );

    /**
     * @return Maximum strength when fully healthy.
     */
    public float getMaxStrength();

    /**
     * @return Current actual strength.
     */
    public float getStrength();

    /**
     * @return Current relative strength as number between 0 and 1.
     */
    public float getRelativeStrength();

    /**
     * @return true if strength = 0
     */
    public boolean isDead();
}
```

LivingCreature

LivingCreature implementiert dieses Interface. Damit können sich später die Monster auch gegenseitig angreifen.

Ein Monster bekommt auch ein Item (attackCapability), quasi als "eingebaute Waffe". Dann muss ich nur noch eine attack() Methode bei Monster implementieren und diese aus dem Gameloop aufrufen:

Monster

```java
public void attack() {
    for ( Field neighbourField : getCurrentField().getNeighbours().values() ) {
        if ( neighbourField != null && neighbourField.getBlocker() instanceof Impactable ) {
            System.out.println( "The monster " + getName() + " attacks!" );
            ((Impactable) neighbourField.getBlocker()).receiveImpact(
                    attackCapability.getImpactOnOthers()
            );
        }
    }
}
```

Game.play()

```java
//...
for ( Monster monster : monsters ) {
    monster.attack();
}
//...
```

15. Refactoring: Stepdown Rule / Keep Methods Small in *Game.play()*

Die *play()* Methode ist jetzt viel zu lang.

```
public void play() {
    System.out.println( "Hello! What is your name?" );
    while ( player.getName() == null ) {
        String name = scanner.nextLine();
        if ( name != null && name.length() > 0 ) {
            player.setName( name );
            System.out.println( "Hello, " + name +
                    ". Your task is to navigate the dungeon and kill the monsters. Good luck!" );
        }
        else {
            System.out.println( "Please tell me your name ..." );
        }
    }

    while( !isFinished() ) {
        dungeon.print();
        System.out.println();
        try {
            Command command = nextCommand();
            player.execute(command);
            if ( player.getCurrentField().containsItem() ) {
                System.out.println( "On the ground, you see a " + player.getCurrentField().getItem() + "." );
            }
            System.out.println();
        }
        catch ( CommandException commandException ) {
            System.out.println( commandException.getMessage() );
            System.out.println( "Please try again.");
        }

        for ( Monster monster : monsters ) {
            monster.attack();
        }
    }
}
```

Ich teile sie in drei Teilmethoden auf.

```
public void play() {
    readPlayerName();

    while( !isFinished() ) {
        dungeon.print();
        System.out.println();

        executePlayerCommand();
        operateMonsters();
    }
}
```

16. Nächstes Feature: Player greift Monster an

Das geht einfach mit den vorhandenen Methoden - nur *Player.use(...)* muss implementiert werden.

Player

```
private void use( String itemName ) throws CommandException {
    if ( !inventory.containsKey( itemName ) )
        throw new CommandException( this + " doesn't have a " + itemName + "." );
    Item item = inventory.get( itemName );

    // impact on self
    receiveImpact( item.getImpactOnSelf() );
    // impact on others
    for ( Field neighbourField : getCurrentField().getNeighbours().values() ) {
        if ( neighbourField != null && neighbourField.getBlocker() instanceof Impactable) {
            Impactable impactable = (Impactable) neighbourField.getBlocker();
            System.out.println( this + " attacks " + impactable + " with a " + itemName + "!" );
            ((Impactable) neighbourField.getBlocker()).receiveImpact( item.getImpactOnOthers() );
        }
    }
}
```

Es gibt noch zwei Kleinigkeiten zu fixen: Tote Monster sollten nicht mehr angreifen 😅, und das heilende Potion wirkt noch nicht.

17. Letztes Feature: Monster bewegen sich

Die Monster sollen sich mit einer gewissen Wahrscheinlichkeit auch bewegen können. Auch das geht mit unserer Struktur jetzt sehr schnell:

Eine neue Zeile in Game.operateMonsters()

```
private void operateMonsters() {
    for ( Monster monster : monsters ) {
        monster.randomWalk();
        monster.attack();
    }
}
```

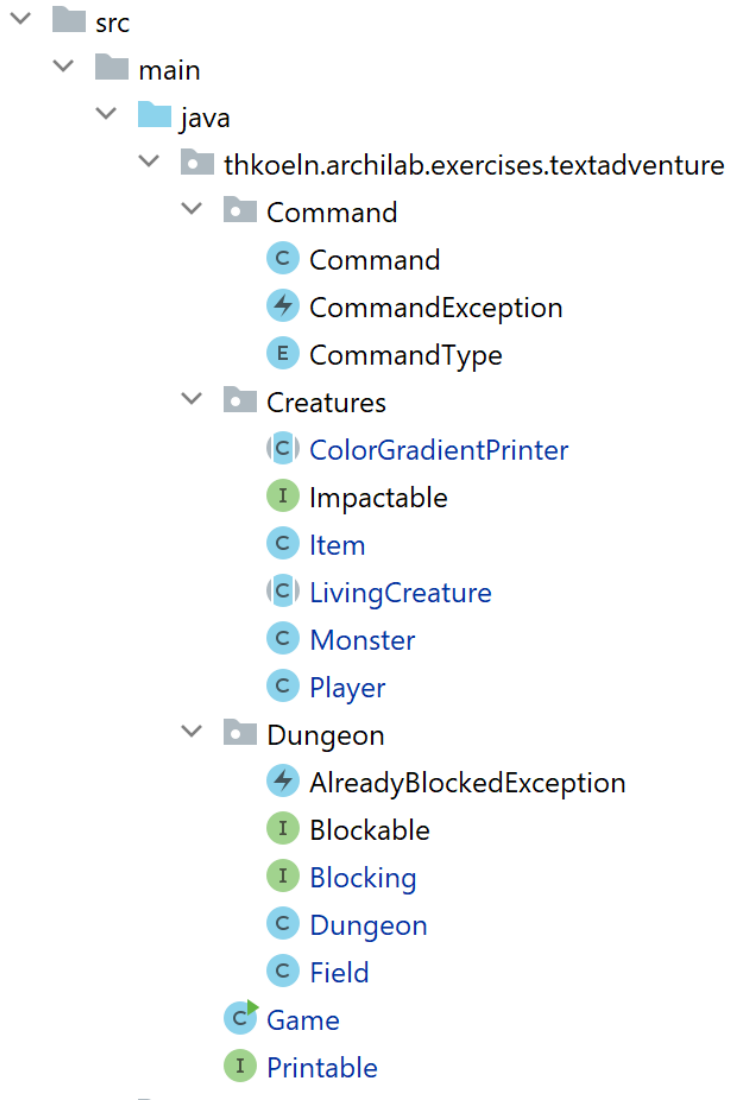
Eine Methode Monster.randomWalk()

```
public void randomWalk() {
    Random random = new Random();
    // move in 7 out of 10 cases
    if ( random.nextFloat() > 0.3f ) {
        // if this random move is illegal, just don't do it :-), no damage done.
        try { move(CommandType.randomDirection()); }
            catch ( CommandException commandException ) {}
    }
}
```

Danach noch ein bisschen Bugfixing - z.B. greifen die Monster momentan auch tote Co-Monster weiter an. That's it!

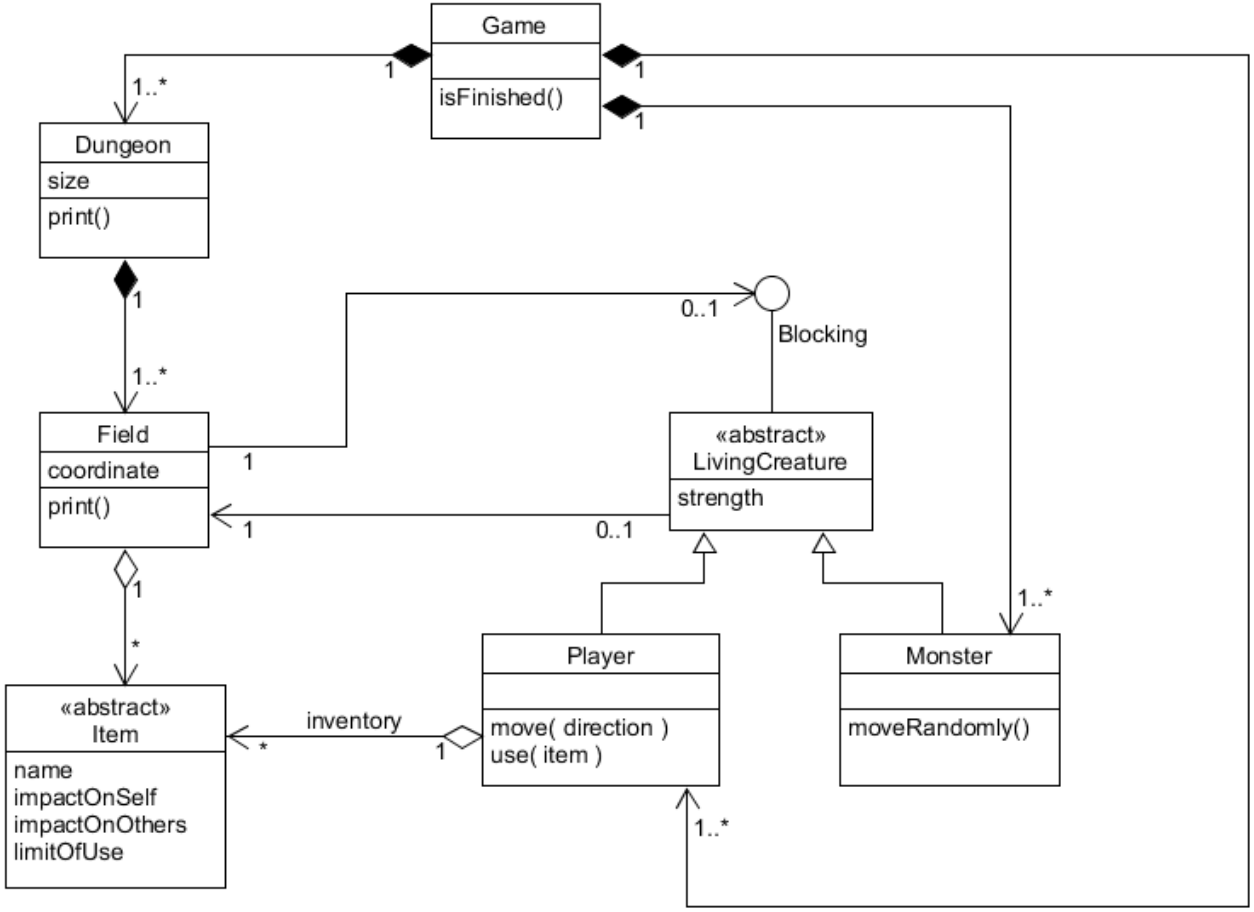
18. Refactoring: Package-Struktur

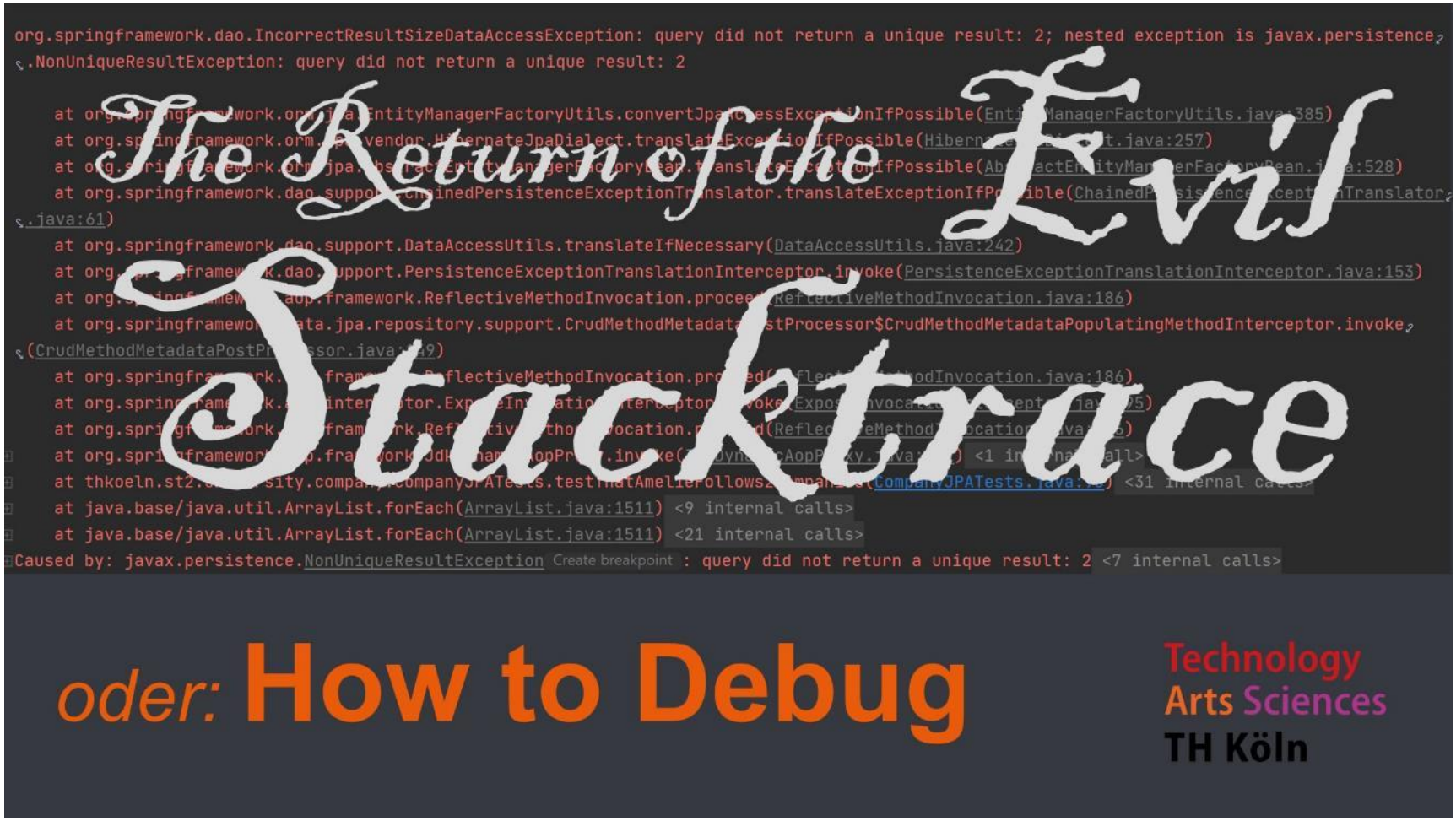
Mittlerweile sind es eine ganze Menge Klassen, Interfaces und Enums (16 Stück) geworden. Daher mache ich als letzten Schritt noch eine (aus meiner Sicht sinnvolle Package-Struktur und verschiebe die Klassen etc. entsprechend.



19. Revisiting the Model

Das Modell habe ich während der Entwicklung nicht mehr angefasst, fand es aber in der Kommunikation anschließend hilfreich. Daher habe ich es noch einmal an den wesentlichen Stellen aktualisiert.


Game
isFinished()
Dungeon
size
print()
Field
coordinate
print()
«abstract»
LivingCreature
strength
Blocking
«abstract»
Item
name
impactOnSelf
impactOnOthers
limitOfUse
inventory
Player
move(direction)
use(item)
Monster
moveRandomly()
1
1..*
0..1
*



https://www.youtube.com/watch?v=vGSNgam_GoE

Wie findet man Fehler?

```
org.springframework.dao.IncorrectResultSizeDataAccessException: query did not return a unique result: 2; nested exception is javax.persistence.NonUniqueResultException: query did not return a unique result: 2

    at org.springframework.orm.jpa.EntityManagerFactoryUtils.convertJpaAccessExceptionIfPossible(EntityManagerFactoryUtils.java:385)
    at org.springframework.orm.jpa.vendor.HibernateJpaDialect.translateExceptionIfPossible(HibernateJpaDialect.java:257)
    at org.springframework.orm.jpa.AbstractEntityManagerFactoryBean.translateExceptionIfPossible(AbstractEntityManagerFactoryBean.java:528)
    at org.springframework.dao.support.ChainedPersistenceExceptionTranslator.translateExceptionIfPossible(ChainedPersistenceExceptionTranslator.java:61)
    at org.springframework.dao.support.DataAccessUtils.translateIfNecessary(DataAccessUtils.java:242)
    at org.springframework.dao.support.PersistenceExceptionTranslationInterceptor.invoke(PersistenceExceptionTranslationInterceptor.java:153)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
    at org.springframework.data.jpa.repository.support.CrudMethodMetadataPostProcessor$CrudMethodMetadataPopulatingMethodInterceptor.invoke(CrudMethodMetadataPostProcessor.java:149)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
    at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:95)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
    at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:212) <1 internal call>
    at thkoeln.st2.university.company.CompanyJPATests.testThatAmelieFollows2Companies(CompanyJPATests.java:93) <31 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <21 internal calls>
Caused by: javax.persistence.NonUniqueResultException Create breakpoint : query did not return a unique result: 2 <7 internal calls>
```

- In diesem Video geht es darum, den Stacktrace richtig zu lesen und den Debugger bedienen zu können, um Fehler im eigenen Code aufspüren zu können.
- Dies spielt das Video an drei Beispielen praktisch durch:

- Text Adventure
  - Die rätselhafte Nullpointer-Exception
- Campus Management System (ST1)
  - Warum ein falscher Status?
  - Hilfe, Spring spinnt

1) Text Adventure - Die rätselhafte Nullpointer-Exception

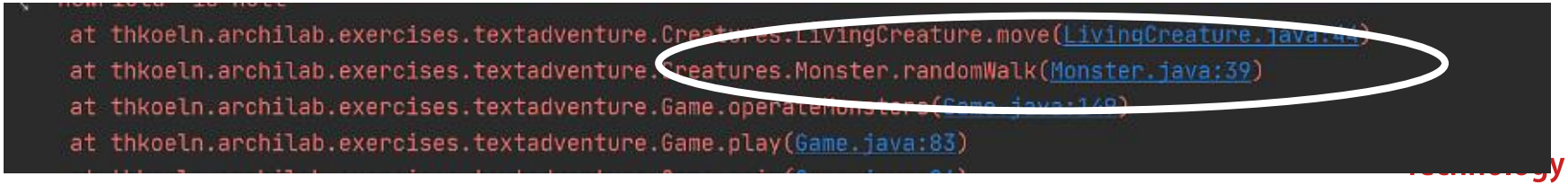
```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "thkoeln.archilab
.exercises.textadventure.Dungeon.Field.block(thkoeln.archilab.exercises.textadventure.Dungeon.Blocking)" because
 "newField" is null
    at thkoeln.archilab.exercises.textadventure.Creatures.LivingCreature.move(LivingCreature.java:44)
    at thkoeln.archilab.exercises.textadventure.Creatures.Monster.randomWalk(Monster.java:39)
    at thkoeln.archilab.exercises.textadventure.Game.operateMonsters(Game.java:149)
    at thkoeln.archilab.exercises.textadventure.Game.play(Game.java:83)
    at thkoeln.archilab.exercises.textadventure.Game.main(Game.java:26)

Process finished with exit code 1
```

- Der Player bewegt sich innerhalb des Dungeons mit legalen Bewegungen – trotzdem fliegt eine Nullpointer-Exception (siehe Stacktrace).
  - Der Stacktrace hat die Fehlerursache in der ersten Zeile
  - nachfolgende Zeilen enthalten die Aufrufhierarchie (in umgekehrter Reihenfolge, also “von unten nach oben”)
- Hier ist der Stacktrace sehr übersichtlich – oft (besonders wenn man ein Framework wie Spring benutzt) ist er viel länger und unübersichtlicher.

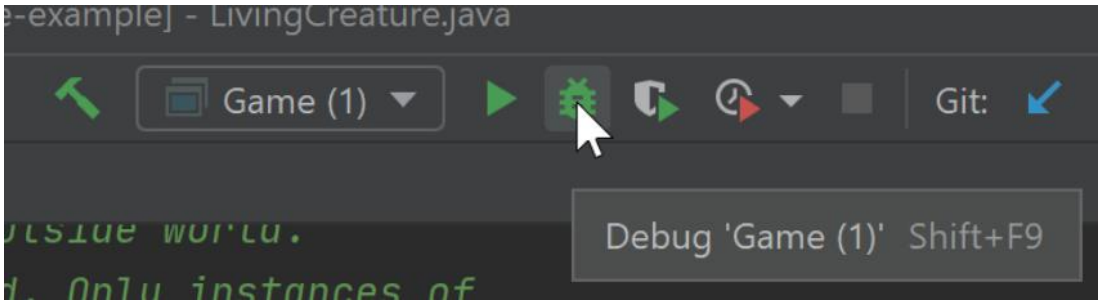
Text Adventure - Die rätselhafte Nullpointer-Exception

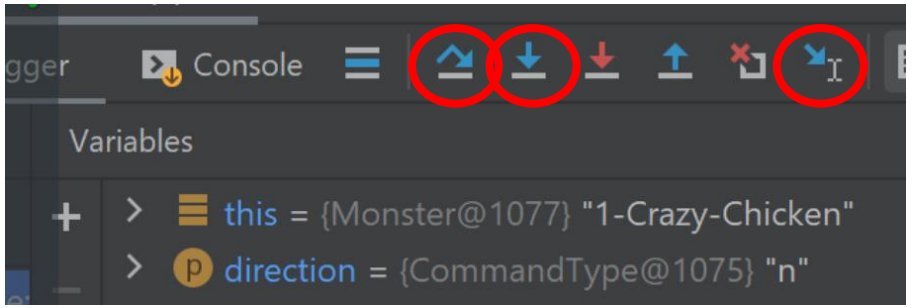
```
protected void move( CommandType direction ) throws CommandException {
   Field newField = currentField.getNeighbours().get( direction );

   if ( newField == null ) throw new CommandException( "You can't go that way!" );

   // The order of the following lines matters! First, blocking the new field is required. If there is
   // already an occupant on the new field, an exception is thrown and we remain safely on our current field.
   // If we would leave the current field first, and then block the new field, we would sit on no field
   // at all!
   newField.block( this );
   currentField.unblock();
   currentField = newField;
}
```

- Mit Hilfe des Debuggers konnte man die Ursache finden: in der Move-Methode fehlte eine Prüfung des “newfield” auf Null. (Siehe rote Umrandung – diese Prüfung war für das Beispiel rausgenommen worden.)
  - “newfield” enthält das Feld, wo die Creature als nächstes hingehen soll.
  - Es ist Null, wenn die Creature auf ein Feld außerhalb des Grids gehen will.
- Der Debugger zeigte auch: Auch die Monster nutzen “move”, wenn sie sich in zufällige Richtungen bewegen. Daher flog die Exception auch, wenn der Player sich korrekt bewegt hatte. In dem Fall hatte ein Monster versucht, einen illegalen Move zu machen.
- Man hätte das auch direkt im Stacktrace sehen können: Siehe zweite Zeile der Aufruf-Hierarchie.



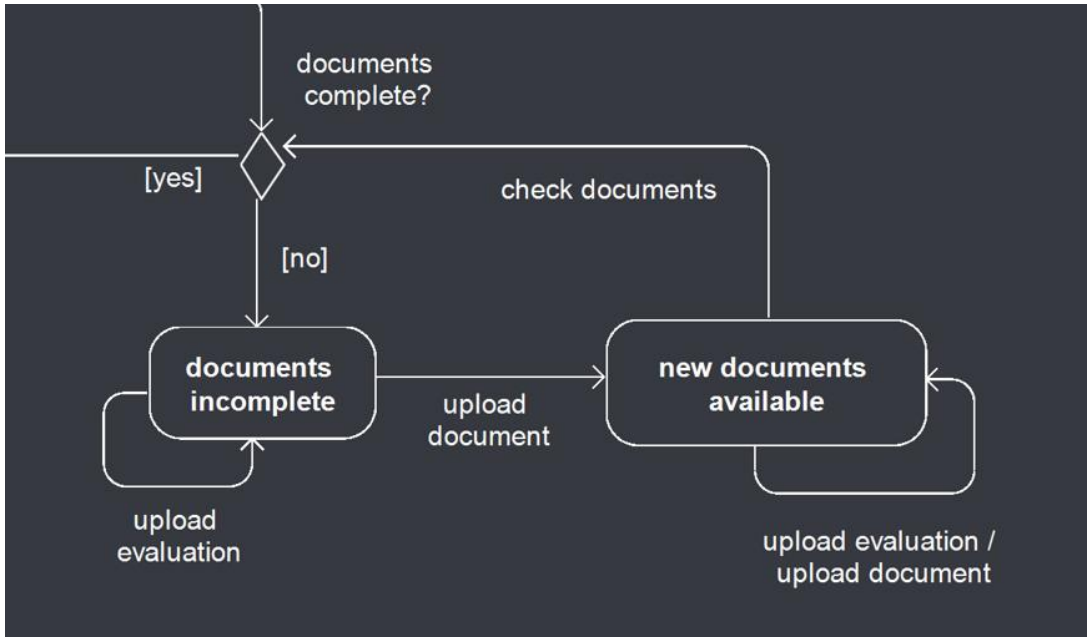
Wie man den Debugger bedient





- In IntelliJ startet man den Debugger via Click auf das “Bug-Symbol”, oder aus dem Kontextmenu (“Debug …”, direkt unter “Run …”) (Bild links)
- Man braucht man drei wesentliche Shortcuts für den Debugger:
- **F7** – Step Into
  - Wenn die aktuelle Zeile einen Methodenaufruf enthält, springe in diese Methode hinein.
- **F8** – Step Over
  - Wenn die aktuelle Zeile einen Methodenaufruf enthält, führe sie aus, gehe aber nicht in den Code (“spring drüber”).
- **Alt + F9** – Run until Cursor
  - Laufe weiter bis zu der Position im Code, wo sich gerade der Cursor befindet.
- Diese Kommandos gibts auch als Icons im Debug-Fenster (Bild rechts)

2) Campus Mgmt System (CAMS) - Warum falscher Status?



Wiederholung aus ST1:

- Wenn die Dokumente schon vollständig sind, geht es nach links weiter (=> siehe nächste Seite).
- Ansonsten ist EnrollmentApplication im Zustand “documents incomplete”.
- Wenn die Bewerberin ein neues Dokument hochlädt (z.B. das Abiturzeugnis), dann geht das Entity in den Zustand “new documents available” über.
  - In diesem Zustand kann die Bewerberin auch noch mehr Dokumente hochladen.
- Erst wenn die Sachbearbeiterin erneut die Dokumente prüft, geht es wieder an den Anfang der Schleife.
- Also: **Sobald ein neues Dokument hochgeladen ist, muss der Zustand auf “new documents available” springen** – als Zeichen für Sachbearbeiterin, dass sie jetzt prüfen muss.

2) CAMS - Warum falscher Status? **Test schlägt fehl**

```
// preparation of state, and test for proper state
enrollmentApplication.uploadResume( resume );
enrollmentApplication.checkDocuments( Boolean.FALSE );
enrollmentApplication.uploadDocument( document1 );

assertEquals( NEW_DOCUMENTS_AVAILABLE, enrollmentApplication.state() );
```

- Das prüft ein Test (oben der Ausschnitt). Nach Upload (dritte Zeile) muss der Zustand *NEW_DOCUMENTS_AVAILABLE* sein (vierte Zeile).
- Der Test schlägt aber fehl: siehe Stacktrace. Der Zustand ist immer noch *DOCUMENTS_INCOMPLETE.*

```
org.opentest4j.AssertionFailedError:
Expected :NEW_DOCUMENTS_AVAILABLE
Actual   :DOCUMENTS_INCOMPLETE
<Click to see difference>

<5 internal calls>
    at thkoeln.st2.university.enrollment.EnrollmentApplicationStatusTests.test_NEW_DOCUMENTS_A
<35 internal calls>
    at java.base/java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:183) <3 in
    at java.base/java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:183) <4 in
```

2) CAMS - Warum falscher Status? **Test schlägt fehl**

- Der Stacktrace hilft hier gar nicht: er zeigt nur den Aufrufstack des Tests.
- Stattdessen muss man mit dem Debugger in die Methode EnrollmentApplication.state().

```
assertEquals( NEW_DOCUMENTS_AVAILABLE, enrollmentApplication.state() );
```

- Geht man mit dem Debugger durch, landet man (wie erwartet) in der Methode “isDocumentCheckNeeded”:

```
public EnrollmentStateType state() {
  if ( matriculationNumber != null ) return ENROLLED;
  if ( paymentReceipt != null ) return FEES_PAID;
  if ( evaluation != null ) {
    return evaluation.getFulFillsRequirement() ? APPLICATION_GRANTED : APPLICATION_DENIED;
  }
  if( documentsComplete != null ) {
    if( documentsComplete ) return EVALUATION_MISSING;
    return isDocumentCheckNeeded() ? NEW_DOCUMENTS_AVAILABLE : DOCUMENTS_INCOMPLETE;
  }
  return ( resume != null ) ? RESUME_UPLOADED : ENROLLMENT_REQUESTED;
}
```

2) CAMS - Warum falscher Status? **Verwende Debugger (1)**

- Der Stacktrace hilft hier gar nicht: er zeigt nur den Aufrufstack des Tests.
- Stattdessen muss man mit dem Debugger in die Methode EnrollmentApplication.state().

```
assertEquals( NEW_DOCUMENTS_AVAILABLE, enrollmentApplication.state() );
```

- Geht man mit dem Debugger durch, landet man (wie erwartet) in der Methode “isDocumentCheckNeeded”:

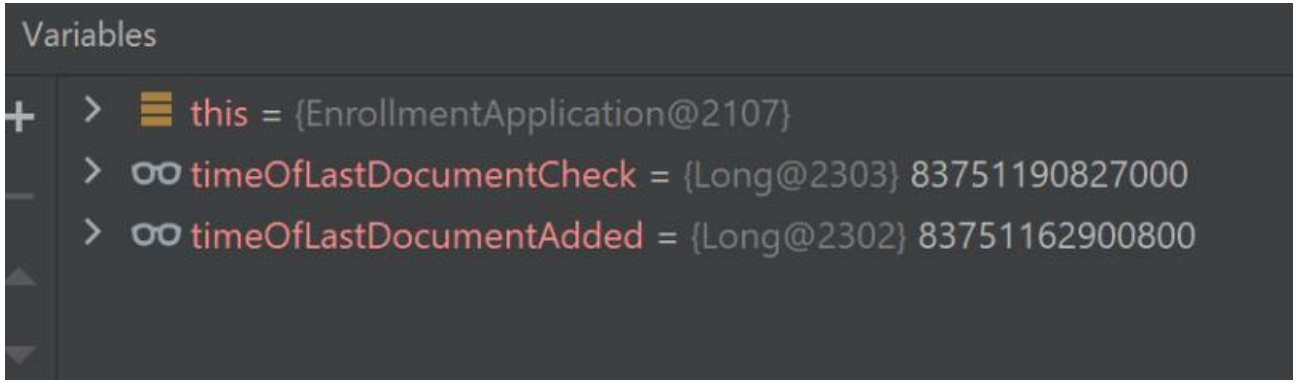
```
public EnrollmentStateType state() {
  if ( matriculationNumber != null ) return ENROLLED;
  if ( paymentReceipt != null ) return FEES_PAID;
  if ( evaluation != null ) {
    return evaluation.getFulFillsRequirement() ? APPLICATION_GRANTED : APPLICATION_DENIED;
  }
  if( documentsComplete != null ) {
    if( documentsComplete ) return EVALUATION_MISSING;
    return isDocumentCheckNeeded() ? NEW_DOCUMENTS_AVAILABLE : DOCUMENTS_INCOMPLETE;
  }
  return ( resume != null ) ? RESUME_UPLOADED : ENROLLMENT_REQUESTED;
}
```

2) CAMS - Warum falscher Status? **Verwende Debugger (2)**

- Hier werden zwei Zeitstempel verglichen, um zu entscheiden, ob schon etwas Neues zum Prüfen da ist.

```
public Boolean isDocumentCheckNeeded() {
  return ( timeOfLastDocumentCheck < timeOfLastDocumentAdded );
}
```

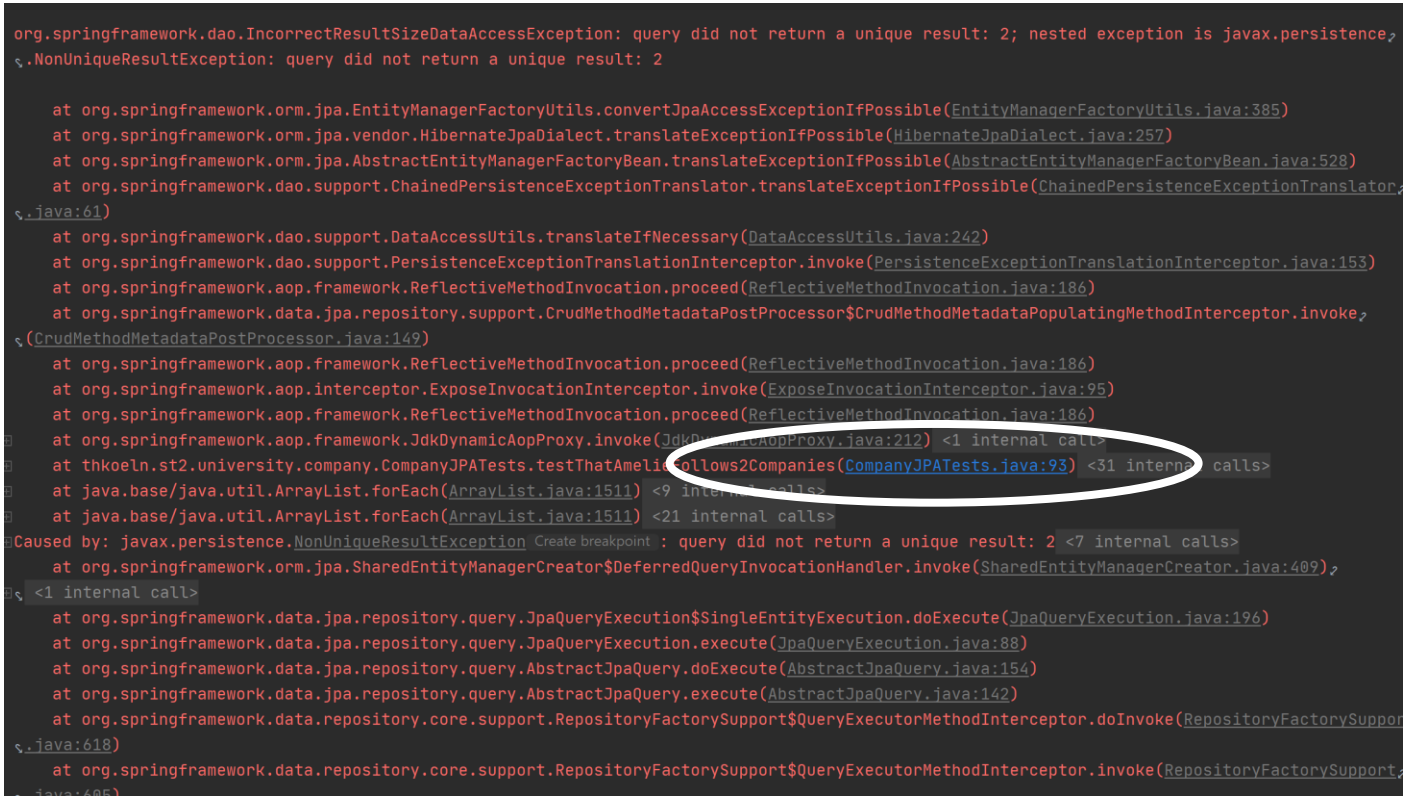
- Im Debugger kann man sich die Variablenwerte anzeigen lassen:



- Damit hat man auch die Fehlerursache identifiziert: Obwohl ein Dokument im Test neu hochladen worden war, war der Zeitstempel timeOfLastDocumentAdded **älter** als der Prüf-Zeitstempel timeOfLastDocumentCheck.
- Damit war klar, dass beim Hochladen das Setzen des Zeitstempels vergessen worden war.

3) CAMS – Hilfe, Spring spinnt!

- In diesem Fall fliegt eine nicht geplante Exception während eines Tests.Der Stacktrace ist lang und unübersichtlich.



- Zuerst sucht man nach der “eigenen” Klasse im Stacktrace (siehe Markierung). Folgende Stelle ist die Ursache:

```
Company followedCompany = companyRepository.findCompaniesByFollowersContaining( amelie );
```

```
Repository extends CrudRepository<Company, UUID> {
mpaniesByFollowersContaining( Student student );
```

3) CAMS – Hilfe, Spring spinnt! – Stackoverflow-Recherche

- Es wird augenscheinlich einfach eine Repo-Methode aufgerufen.
- Weder der Debugger noch der Aufruf-Stack helfen hier weiter.
- Stattdessen ist es sinnvoll, sich die Exception einmal genauer anzuschauen:

```
org.springframework.dao.IncorrectResultSizeDataAccessException: query did not return a unique resul
.NonUniqueResultException: query did not return a unique result: 2
```

- Hier empfiehlt sich nach dieser Exception zu googlen, insbesondere bei Stackoverflow. Dort findet sich eine Frage eines Nutzers (natürlich mit anderen Entities), aber mit einem ähnlichen Problem. Die Antwort in Stackoverflow gibt den entscheidenden Hinweis:

> It seems like there are multiple Users in your DB-Table with the same username. So `User findByUsername(String username);` returns more than one Result.
>
> You could do one of the following things:
>
> 1. Make the username-Column in your DB unique.
> 2. Change your repository method to `List<User> findByUsername(String username);` to get all Users with that username.

3) CAMS – Hilfe, Spring spinnt! – falsche Repo-Methode

- Wenn wir in dem Repo CompanyRepository nachschauen, finden wir dort die verwendete Finder-Methode:

```
public interface CompanyRepository extends CrudRepository<Company, UUID> {
   public Company findCompaniesByFollowersContaining( Student student );
}
```

- Wenden wir den Hinweis aus Stackoverflow an, dann wird deutlich, dass “Company – Student” eine n:m-Beziehung ist und findCompaniesByFollowersContaining(Student student) damit mehr als eine Company zurückgeben kann.
- Also müsste die Finder-Methode so aussehen:

```
public interface CompanyRepository extends CrudRepository<Company, UUID> {
   public List<Company> findCompaniesByFollowersContaining( Student student );
}
```

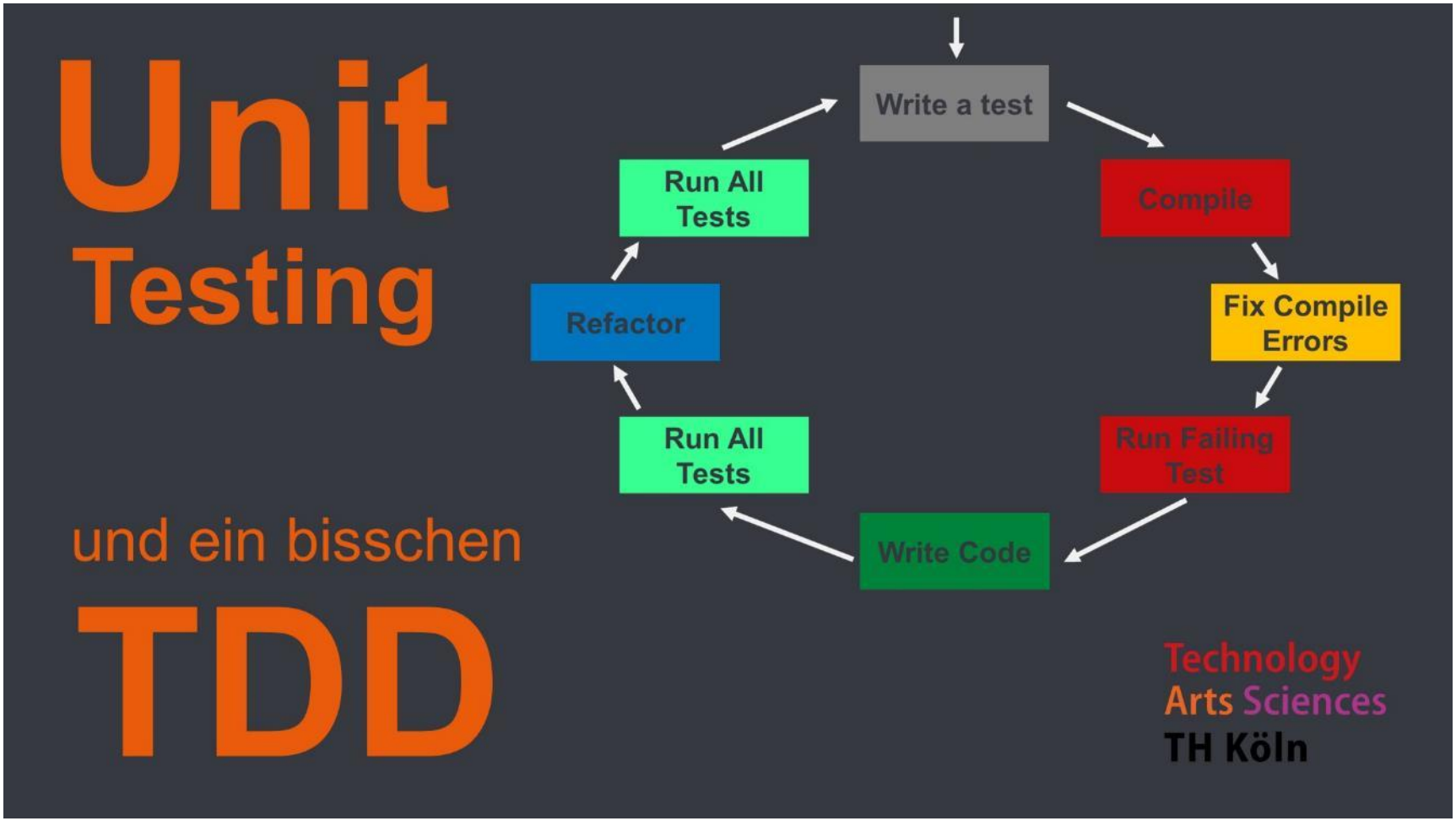
- Das erklärt dann auch den Text der Exception:

```
org.springframework.dao.IncorrectResultSizeDataAccessException: query did not return a unique resul
s.NonUniqueResultException: query did not return a unique result: 2
```

- Wenn man Aufrufstelle auch noch entsprechend ändert, hat man das Problem gelöst.

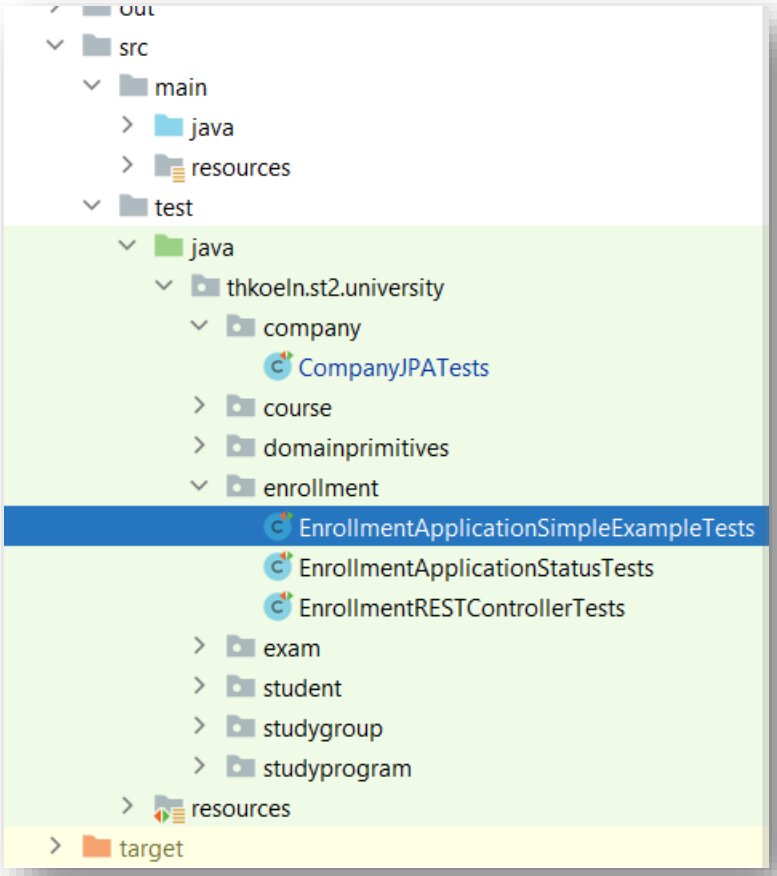
Zusammenfassung: **Drei Strategien der Fehlersuche**

- Text Adventure - Die rätselhafte Nullpointer-Exception
  - Stacktrace nutzen, um Problemstelle zu finden
  - Debugger, um zu verstehen, **wann genau** das Problem auftritt
  - **Fürs nächste Mal:** Stacktrace genau anschauen – auch den Aufrufer der problematischen Methode.
- Campus Management System - Warum ein falscher Status?
  - Stacktrace hilft nicht – direkt mit Debugger prüfen, was passiert
  - Variableninhalte prüfen und versuchen, “ungewöhnliche Dinge” zu entdecken
  - Daraus kommt dann oft die Fehler-Einsicht
- CAMS - Hilfe, Spring spinnt
  - Weder Stacktrace noch Debugger bieten sinnvolle Hilfe
  - Aber: die Exception ist hilfreich. Googeln bringt den entscheidenden Hinweise (Stackoverflow)
  - **Fürs nächste Mal:** Exception-Text genau anschauen – oft wird daraus (mit etwas Nachdenken) schon klar, woran es lag.



https://www.youtube.com/watch?v=Y1xXZWdtbgY

Ein ganz simples Test-Beispiel




```java
package thkoeln.st2.university.enrollment;


import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class EnrollmentApplicationSimpleExampleTests {
   @Test
   public void testNothingReally() {
      assertTrue( 1==1 );
   }
}
```

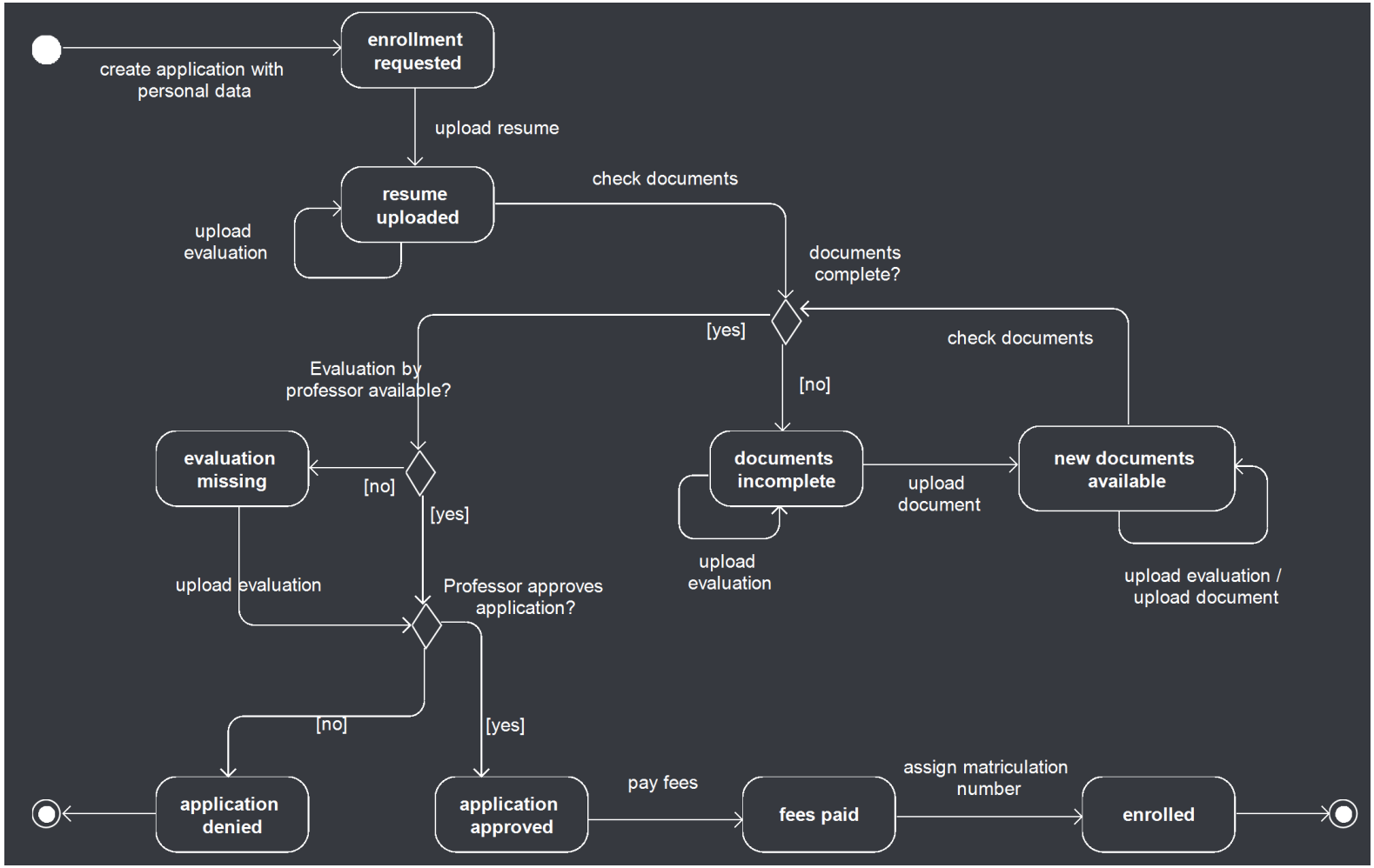
- Tests werden in den „test“ Zweig der Sourcecode-Folder (unter „src“, neben „main“) gelegt.
- Tests sind erstmal einfache Java-Klassen. Ein Test wird mit @Test annotiert.
- Dieser Test hier macht gar nichts ☹ - er prüft nur, ob 1 = 1.

org.junit.jupiter.api.Assertions

- Die assert… Funktionen können in den Tests verwendet werden, um erwartete Ergebnisse zu prüfen
  - assertNull(…) + assertNotNull(…)
    - Prüfung auf Null / not Null
  - assertEquals(…)
    - Prüfung, ob Ergebnis gleich Erwartung (Aufruf von equals(…))
  - assertSame(…) + assertNotSame(…)
    - Prüfung auf Identität (Aufruf von “==“)
  - assertArrayEquals(…)
    - Gleichheit zweier Arrays
  - assertTrue(…) + assertFalse(…)
    - Prüfung, ob eine Bedingung sich zu “True” oder “False” auswertet
  - assertThrows(…)
    - Prüfung, ob eine erwartete Exception geworfen wird
    - Anwendung wie folgt:

```
assertThrows( DungeonException.class, () ->
   { dungeon =  new Dungeon( -1, 0 ); }
);
```

CAMS (nicht im Video): Statusfolge „Enrollment Application“



- Wir nehmen noch einmal die Statusfolge zu „Enrollment Application“ als Beispiel für das Schreiben von Tests.

CAMS (nicht im Video)**: Test der Statusfolge „Enrollment Application“**

```
public void setUp() {
   // we could also have a more sophisticated creation of test data ...
   enrollmentApplication = new EnrollmentApplication();
}

@AfterEach
public void tearDown() {
   // not really needed, of course ... had we persistent data, we could clean
   // the database here.
   enrollmentApplication = null;
}
@Test
public void testInitialStateIsCorrect() {
   assertEquals( enrollmentApplication.state(), ENROLLMENT_REQUESTED );
}

@Test
public void testStateIsCorrectAfterResumeUpload() {
   enrollmentApplication.uploadResume( new Document( "my Resume" ) );
   assertEquals( enrollmentApplication.state(), RESUME_UPLOADED );
}

@Test
public void testSecondResumeUploadCausesException() {
   enrollmentApplication.uploadResume( new Document( "my Resume" ) );
   assertThrows( NotAllowedOperationException.class, () ->
      { enrollmentApplication.uploadResume( new Document("another Resume") ); } );
}
```

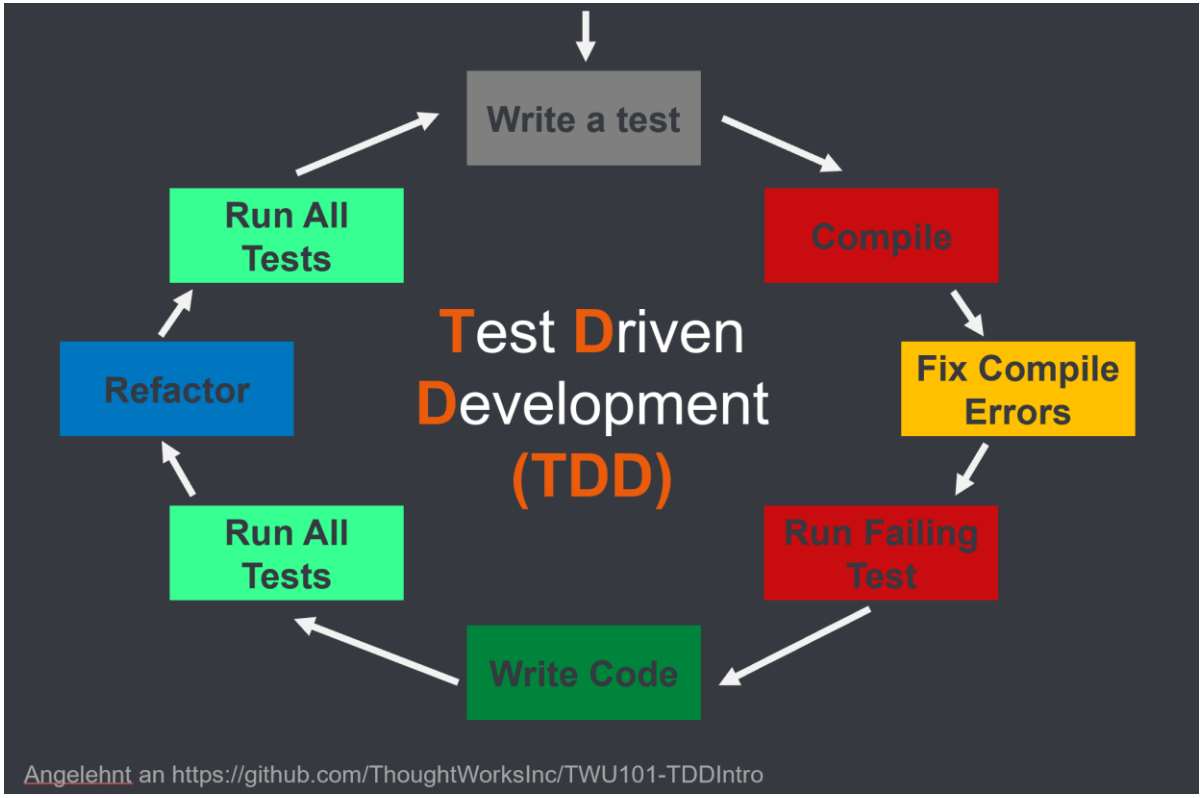
- @BeforeEach, @AfterEach werden vor/nach jedem Test ausgeführt.
- Hier wird geprüft, ob der initiale Zustand richtig ist (also bevor irgendetwas passiert)
- Hier wird getestet, ob nach dem Upload eines Resumes der Zustand entsprechend ist.
- Nach dem Upload eines Resumes darf der Upload kein zweites Mal erfolgen (die Zustandsfolge schreibt das vor). Dieser Test prüft, ob dann eine (erwartete) Exception geworfen wird.

Given – When – Then

- The **given** part describes the state of the world before you begin the behavior you're specifying in this scenario. You can think of it as the pre-conditions to the test.
- The **when** section is that behavior that you're specifying.
- Finally the **then** section describes the changes you expect due to the specified behavior.

```
@Test
public void testEvaluationMissingAfterDocCheck() {
   // given
   enrollmentApplication.uploadResume( new Document( "my Resume" ) );

   // when
   enrollmentApplication.checkDocuments( true );

   // then
   assertEquals( enrollmentApplication.state(), EVALUATION_MISSING );
}
```

Test-Driven Development (TDD)



- TDD ist ein Vorgehen in der Software-Entwicklung, das Unittests aktiv in die Entwicklung einbezieht (statt sie einfach nach der Programmierung hinzuzufügen)

1. Wenn man ein neues Feature entwickeln will, **schreibt man zunächst einen Test**, der das neue Feature voraussetzt.
2. In diesem Test werden Klassen vorkommen, die es noch nicht gibt. Daher **kompiliert** der Code nicht.
3. Also **fixt man zunächst die Compile-Fehler**: z.B. indem man leere Klassen anlegt.
4. Man lässt seinen **Test laufen**: Er wird scheitern ("rot sein"), weil das Feature ja noch nicht da ist (sondern nur leere Klassen)
5. Dann **schreibt man den Code** für sein neues Features.
6. Man lässt immer wieder seine **Tests laufen**. Irgendwann ist alles grün => man ist fertig!
7. Jetzt kann man noch ein bisschen „aufräumen" und **Refactoring** von technischer Schuld betreiben.
8. Danach immer wieder die **Tests laufen** lassen. Wenn alles grün bleibt, hat man durch das Refactoring nichts „verschlimmbessert" oder kaputt gemacht.

TDD am Beispiel von „DungeonTest“

```java
@Test
public void testInvalidDungeon() {
   // given
   // when
   // then
   assertThrows( DungeonException.class, () ->
      { dungeon =  new Dungeon( -1, 0 ); } );
}
```

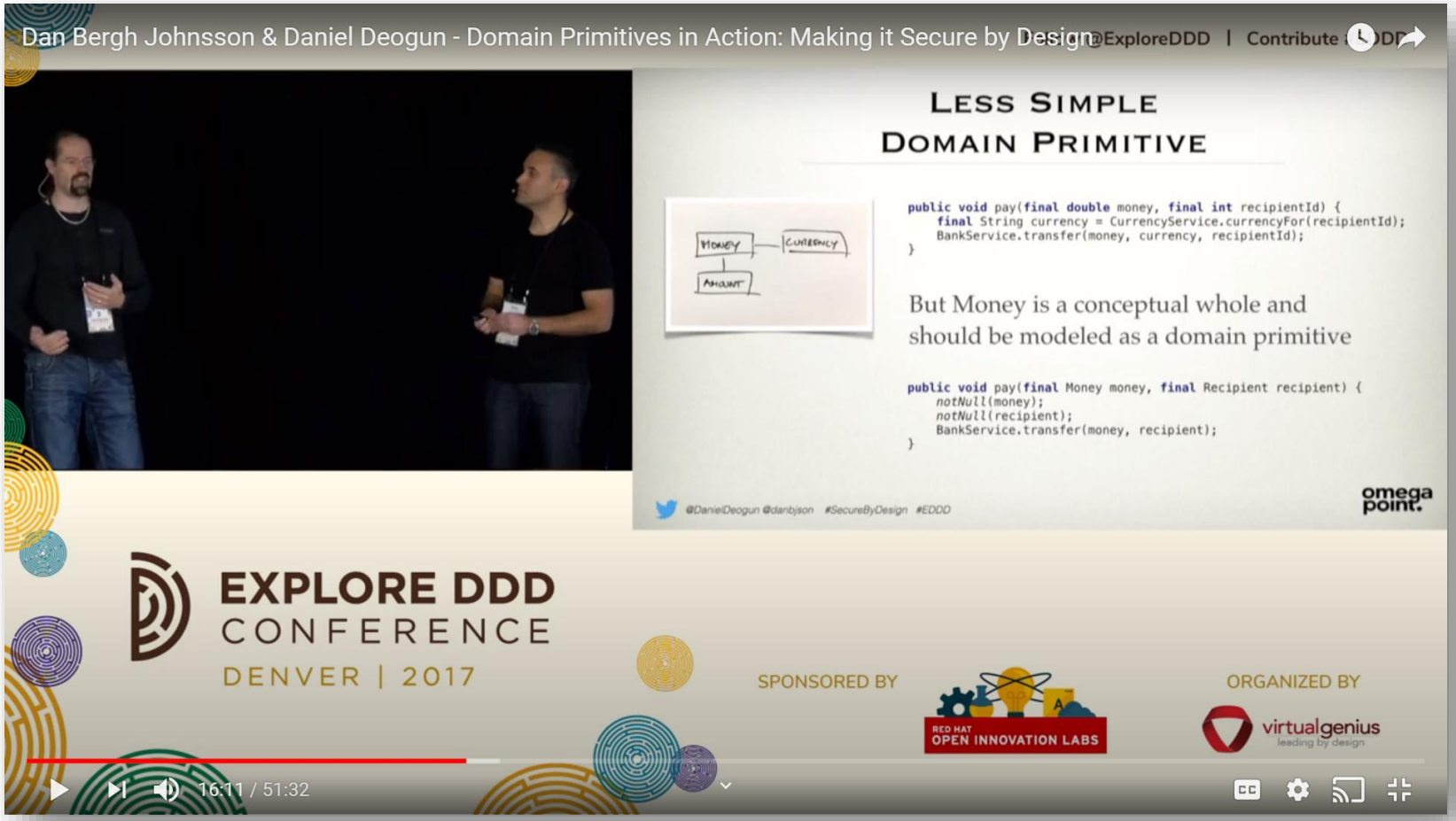
```java
public Dungeon( int width, int height ) {
   if( width <= 0 ) throw new DungeonException( "width must be > 0!" );
   if( height <= 0 ) throw new DungeonException( "height must be > 0!" );
   this.width = width;
   this.height = height;
   …
}
```

- Im Video wird dieses Vorgehen anhand der Methode `testInvalidDungeon` durchgespielt (siehe **links**)
  - (gehört zum Text Adventure, siehe Repo https://git.st.archi-lab.io/students/st2/ss21/exercises/textadventure-v2)
- Hier ist das neue Feature, das entwickelt werden soll, eine Exception bei Initialisierung eines ungültigen Dungeons (Breite von -1). Im existierenden Code gab es diese Exception zunächst nicht; wenn man `new Dungeon( -1, 0 );` aufruft, kommt eine ArrayOutBoundsException.

1. Zuerst wird der obige Test geschrieben.
2. Der Code compiliert anschließend nicht – die geforderte `DungeonException` gibt es noch nicht.
3. Der Compile-Fehler wird dadurch gefixt, dass die Exception einfach mal angelegt wird – sie wird aber noch nicht in den Code eingebaut.
4. Laufenlassen des Tests ergibt „rot“: Die erwartete Exception fliegt (natürlich!) noch nicht.
5. Jetzt wird das Feature wirklich im Konstruktor von Dungeon implementiert (siehe **rechts**). Ist ganz einfach in diesem Fall, nur der Test auf Breite und Höhe > 0.
6. Dann läuft der Test durch und ist grün.
7. *(Refactoring haben wir hier nicht gebraucht …)*

https://www.youtube.com/watch?v=DQrqVkgCDaM

Motivation für *Domain Primitives*



- Dan Bergh Johnsson & Daniel Deogun - Domain Primitives in Action: Making it Secure by Design
- https://www.youtube.com/watch?v=ogjOKlXHi08
- (12:40 – 16:10: Money Example)

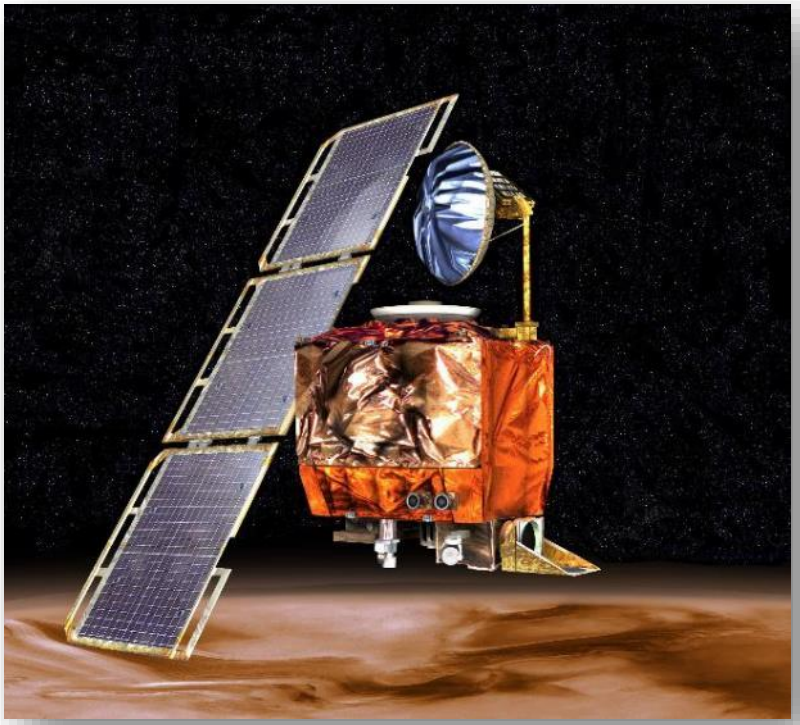
Zwei prominente (sehr teure) Fehler mit Basistypen

Ariadne-5-Absturz, 1996

“However, problems began to occur when the software attempted to stuff this 64-bit variable (…) into a 16-bit integer. (…) For the first few seconds of flight, the rocket’s acceleration was low, so the conversion between these two values was successful. However, as the rocket’s velocity increased, the 64-bit variable exceeded 65k, and became too large to fit in a 16-bit variable.“

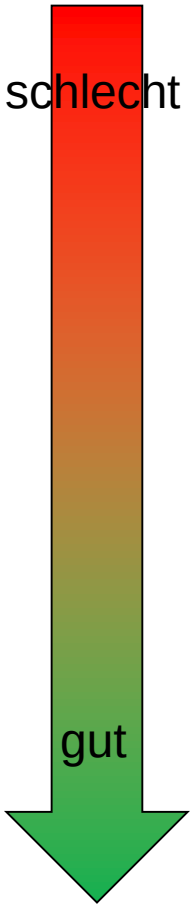
Absturz Mars Climate Orbiter, 1999

“Quality Assurance had not found the use of an imperial unit in external software, despite the fact that NASA’s coding standards at the time mandated use of metric units..“



Quellen: https://www.bugsnag.com/blog/bug-day-mars-climate-orbiter , https://www.bugsnag.com/blog/bug-day-ariane-5-disaster,
Bilder: NASA/JPL/Corby Waste - Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=390903,
DLR German Aerospace Center - Raumfrachter ATV-4 "Albert Einstein" Ariane 5ES Rollout_4, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26533952

Die Rangfolge von „am schlechtesten“ bis „am besten“



1. ~~Attribute als primitive Datentypen (int, float, double, …)~~

Sollte man immer vermeiden (ist ja auch leicht umzusetzen)

2. Attribute als Wrapperklassen (Integer, Float, Double, …)

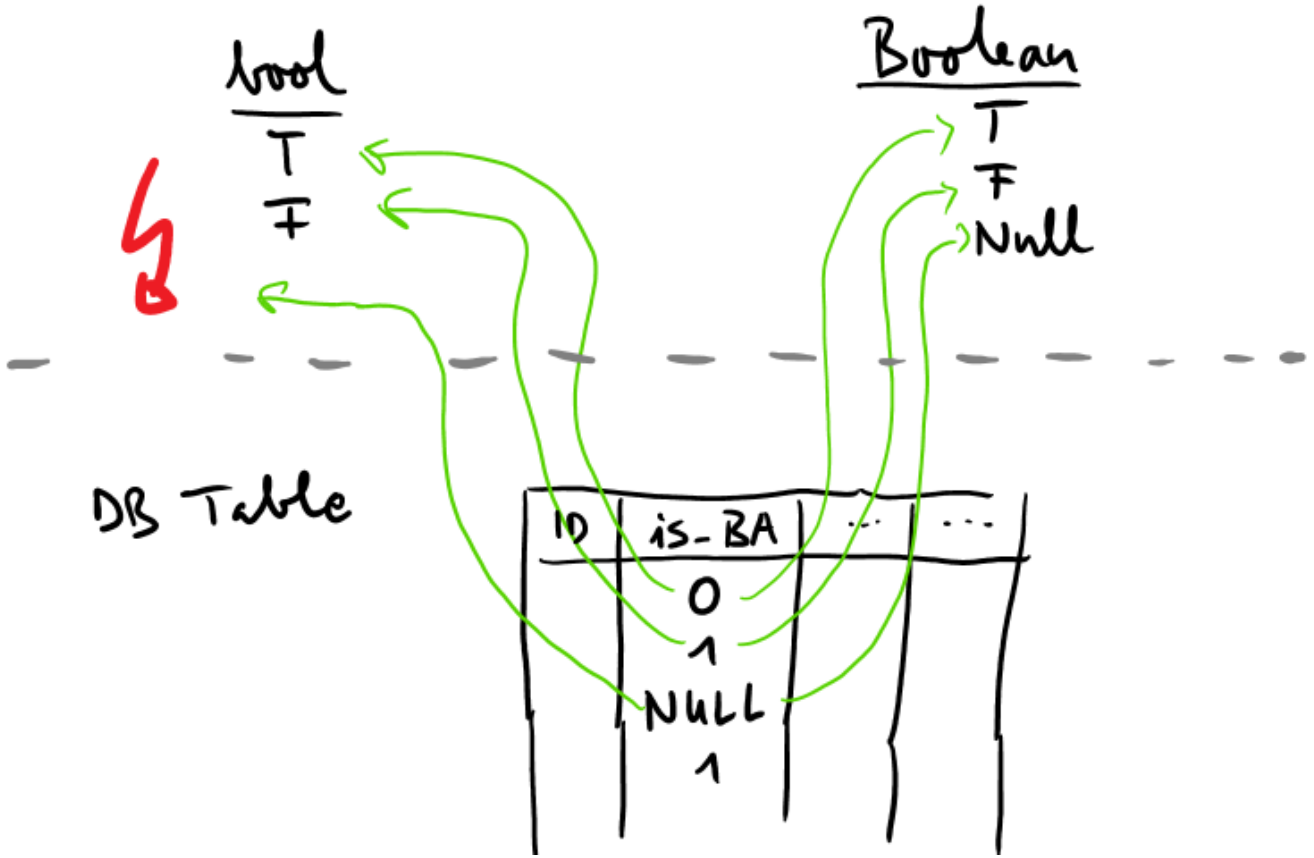
3. Attribute als Wrapperklassen mit domänen-spezifischer Validierung

4. Attribute als „Domain Primitive“ Value Objects
   - Erlaubt Builder / Konverter, spezifische Validations / Exceptions, domänenspezische Funktionalität

5. ~~Attribute als native *Domain Primitives*~~
   - Voller IDE-Support, so wie native Datentypen, aber domänen-spezifisches Verhalten

zumindest in Java leider nicht ver-fügbar, ggfs. in moderneren Sprachen

Option 1) Warum keine „primitive data types“ in Entities?



- Bei einer NULLABLE DB Column kann eine NULL nicht auf einen primitiven Datentypen abgebildet werden, auf einen Wrapper-Typen aber schon (siehe Beispiel).
- Generell sind Frameworks wie Spring eher auf den Umgang mit Objekten ausgelegt; die Wrapperklassen sind damit konsistenter zu benutzen.

Option 3) Wrapperklasse + JPA-Validierung

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

Student

```
@DecimalMin(value = "10000000")
@DecimalMax(value = "99999999")
private Long matrNum;
```

Test

```
@Test
public void testValidation() {
   Student s = new Student();
   s.setName("Hans");
   s.setMatrNum(1234567L);
   Exception ex = assertThrows(TransactionException.class, () -> {
      studentRepository.save(s);
   });
   s.setMatrNum(10000000L);
}
```

- Unsere Domänenregel sagt:
  „Eine Matrikelnummer beginnt mit 1-9 und ist genau 8-stellig“ (*)

- Wertebereich von Attributen kann durch Annotationen wie `@DecimalMin(value = "10000000")` eingeschränkt werden. Es gibt auch noch weitere Annotationen zur Validierung. Damit kann zumindest verhindert werden, dass Entities mit Werten erzeugt werden, die in der Domäne keinen Sinn ergeben.

(*) Ich habe nicht recherchiert, ob diese Regel für die TH Köln genau so stimmt – nehmen wir es einfach mal an.

Definition „Domain Primitive“

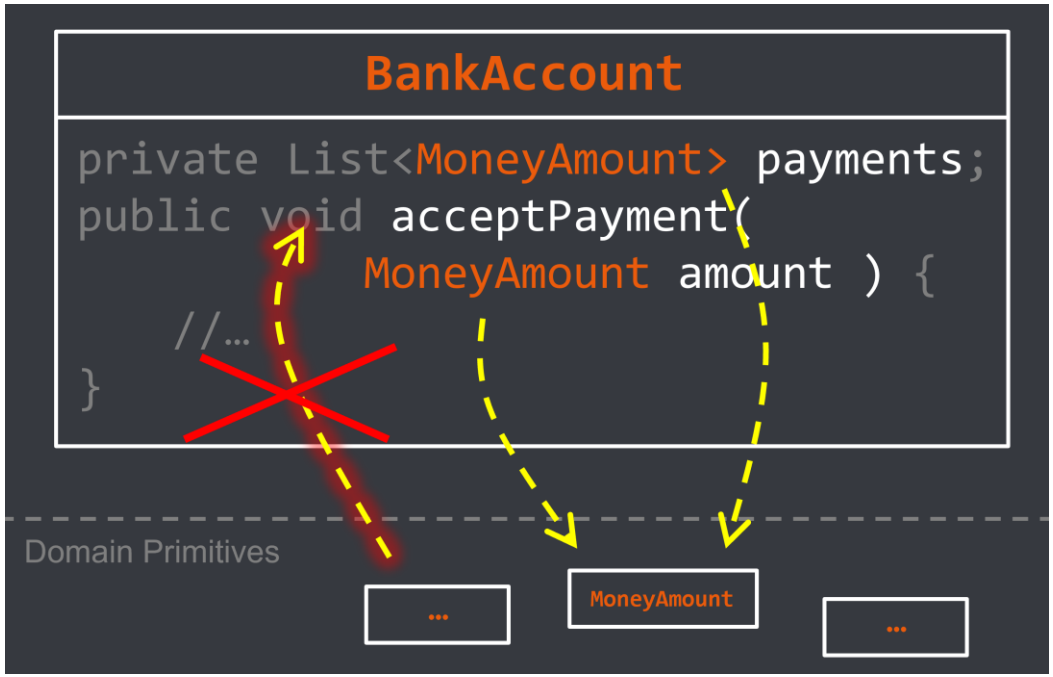
- A value object precise enough in its definition that it, by its mere existence, manifests its validity is called a **domain primitive**.
  - *Robert Clark (2019), https://medium.com/@robot88/domain-primitives-swift-and-you-cb9752c44878*
- […] we require invariants to exist and they must be enforced at the point of creation.
  - *D. B. Johnsson, D. Deogun, D. Sawano (2018), https://freecontent.manning.com/domain-primitives-what-they-are-and-how-you-can-use-them-to-make-more-secure-software/ [Johnsson, Deogun, Sawano]*

- Eigenschaften von Domain Primitives *[Johnsson, Deogun, Sawano]*
  - Their invariants are checked at the time of creation.
  - They can only exist if they’re valid.
  - They should always be used instead of language primitives or generic types.
  - Their meaning’s defined within the boundaries of the current domain, even if the same term exists outside of the current domain.

Regeln für Domain Primitives

1. Value Object
   - keine Setter
   - Änderungen geben neues Objekt zurück (immutable)
2. Erzeugung über Factory-Methode
   - protected oder private Constructor
   - valueOf(…), fromXXX(…)
     - Damit kann man die Semantik der Factory-Methoden besser deutlich machen als über Konstruktoren.
3. Exceptions für Fehler bei Erzeugung
4. Keine Rückreferenzen auf Domain-Entities

Wieso keine Rückreferenzen auf Domain-Entities?



- Eine Referenz kann durch eine Member-Variable eines Typs entstehen, aber auch schon dadurch, dass eine Methode einen Parameter eines Typs hat.
  - Im obigen Beispiel: die Domain Primitive „ MoneyAmount“ kommt im Entity „BankAccount“ 2x vor, einmal als Member, einmal als Parameter.
- Das ist auch völlig ok: Nach dem Depency Inversion Principle soll ich in meinem Code keine volatilen (sich oft ändernden) Konzepte referenzieren. Aber Domain Primitives bilden „Basiskonzepte“ meiner Domäne ab, die werden sich nicht oft ändern. (Oder denken Sie, dass sich der Begriff und die Regeln für „Geldbetrag“ oft ändern?)
- Nur eine Rückreferenz darf auf keinen Fall sein: Dann nämlich ist meine Domain Primitive plötzlich von „BankAccount“ abhängig. Und dann kann ich nicht mehr einfach überall in meinem „MoneyAmount“ benutzen, ohne vorher drüber nachzudenken.

Beispiel für Domain Primitives (1): **Strength / Impact**

Text Adventure *(dieses Beispiel wurde im Video gezeigt)*

Strength (Teil 1)

```
public class Strength {
   private Float max;
   private Float current;


   public static Strength fromMax( Float maxStrength ) { return new Strength( maxStrength ); }

   public static Strength fromCurrentAndMax( Float current, Float maxStrength ) {
      Strength strength = new Strength( maxStrength );
      strength.setCurrent( current );
      return strength;
   }

   public Strength afterImpact( Impact impact ) {
      if ( impact.isNone() ) {
         return this;
      }
      else {
         float newCurrent = this.current + impact.effectOnStrength();
         if ( newCurrent > max ) newCurrent = max;
         if ( newCurrent <= 0f ) newCurrent = 0f;
         return Strength.fromCurrentAndMax( newCurrent, max );
      }
   }

   public Float relative() { return current / max;  }
   public boolean meansDeath() { return current <= 0f;  }
   //...
```

- In dem Beispiel sieht man gut die typischen Eigenschaften einer Domain Primitive:

1. Der „Kontext“ der maximalen Stärke wird zusammen mit der aktuellen Stärke gespeichert. Dadurch kann man immer eine relative Stärke ausrechnen (z.B. für farbliche Darstellung).
2. Es ist ein Value Object, es gibt keine Setter.
3. Es gibt verschiedene Factory-Methoden: üblicherweise wird eine Strength nur über die maximale Stärke initialisiert, aber es gibt auch eine Methode, um eine „geschwächte Stärke“ (wo current < max) zu erzeugen, z.B. zu Testzwecken.
4. „afterImpact“ nimmt einen Impact und wendet ihn auf die Strength an. Als Ergebnis wird eine neue Strength zurückgegeben (Value Object, Immutable)

Beispiel für Domain Primitives (1): **Strength / Impact**

Strength (Teil 2)

```
public class Strength {
  ...

  @Override
  public String toString() {
    return String.format("%.01f", current) + " out of " + String.format("%.01f", max);
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Strength strength = (Strength) o;
    return Float.compare(strength.max, max) == 0 &&
           Float.compare(strength.current, current) == 0;
  }

  protected Strength( Float maxStrength ) {
    if ( maxStrength <= 0f ) throw new StrengthException( "Maximum strength must be > 0" );
    this.max = maxStrength;
    current = maxStrength;
  }

  protected void setCurrent( Float current ) {
    if ( current < 0f ) throw new StrengthException( "Current strength must be >= 0" );
    if ( current > this.max ) throw new StrengthException( "Current strength must be < max" );
    this.current = current;
  }
}
```

5. Man kann toString und equals sinnvoll überschreiben.
6. Der „protected“ Konstruktor wirft eine spezielle StrengthException, wenn man versucht, eine maximale Stärke <=0 zu initialisieren.

Beispiel für Domain Primitives (1): **Strength / Impact**

Impact

```
public class Impact {
   private Float effectOnStrength;

   protected Impact( Float effectOnStrength ) {
      this.effectOnStrength = effectOnStrength;
   }

   public static Impact from( Float effectOnStrength ) {
      return new Impact( effectOnStrength );
   }

   public boolean isNone() {
      return Float.compare(effectOnStrength, 0f) == 0;
   }

   /**
    * Visibility "package" => only to be used by Strength.receiveImpact( impact )
    * @return changeToStrength
    */
   Float effectOnStrength() {
      return effectOnStrength;
   }

   @Override
   public String toString() {
      return String.format("%.01f", effectOnStrength);
   }
}
```

- Auch Impact wird konsequent als Value Object (nicht als Float) implementiert.
- Damit hält man sich z.B. für später offen, mal eine andere Art von Impact zu haben – nicht nur so etwas wie „-10“, sondern auch „halbiere die Stärke“. So eine Zusatzinformation kann man in dem Value Object gut unterbringen.

Beispiel für Domain Primitives (2): **MatrNumber**

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

Student

```
@Getter
@Setter
//@DecimalMin(value = "10000000")
//@DecimalMax(value = "99999999")
//private Long matrNum;
@Embedded
private MatrNumber matrNum;
```

StudentRepository

```
//List<Student> findByMatrNumLessThan( Long matrNum );
List<Student> findByMatrNumLessThan( MatrNumber matrNum );
```

CreateSampleData

```
//s1.setMatrNum( 12345678L );
s1.setMatrNum( MatrNumber.fromLong( 12345678L ) );
```

- Die bessere (aber auch aufwändigere) und „DDD-konformere“ Lösung ist es, *nur* Value Objects als Attribute zuzulassen. Das hat den Vorteil, dass ich neben einer maßgeschneiderten Validierung auch Builder-Methoden und sonstige Domänen-spezifische Logik unterbringen kann.
- **Achtung:** Dieses Vorgehen ergibt nur dann Sinn, wenn man systematisch eine Bibliothek von „Domänenspezifischen atomaren Value Objects“ aufbaut und diese konsequent wiederverwendet. Für jedes Entity neue Value Objects einfach um Integer, String etc. „herum zu wrappen“ ist nutzlos und kontraproduktiv (nur Aufwand, kein Nutzen).

Beispiel für Domain Primitives (2): **MatrNumber**

Beispiel aus dem Campus-Management-System *(kam im Video nicht vor)*

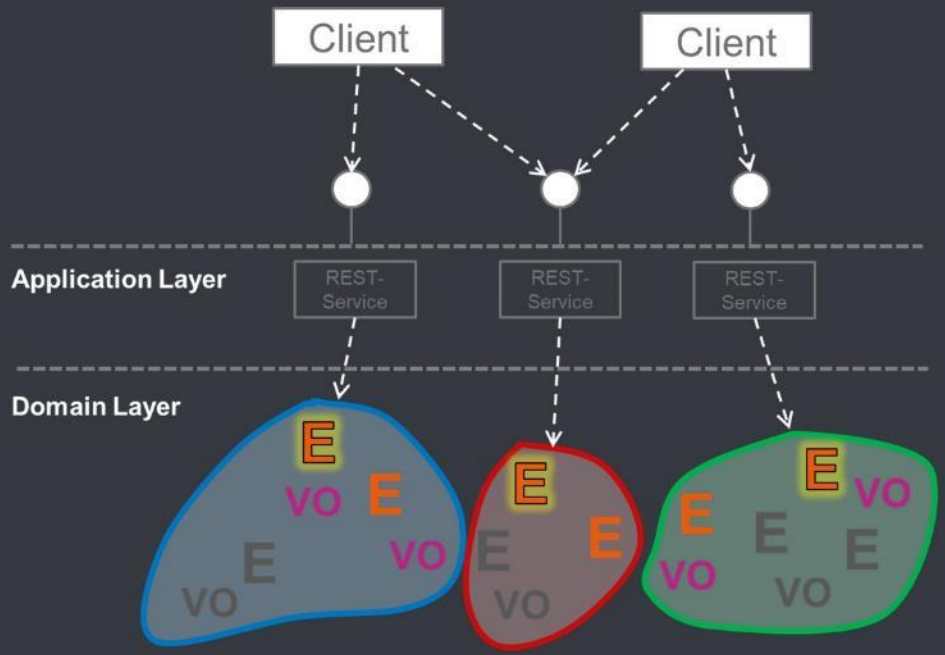
MatrNumber

```
package thkoeln.st2.domainprimitives;
import …

@ToString
@EqualsAndHashCode
@Embeddable
public class MatrNumber {

   @DecimalMin(value = "10000000")
   @DecimalMax(value = "99999999")
   Long num;

   private MatrNumber( Long l ) {
      if (l > 99999999L || l < 1000000L ) {
         throw new MatrNumberException( "invalid number" + l );
      }
      num = l;
   }
   protected MatrNumber() {}

   // a number of builder methods
   public static MatrNumber fromLong( Long l ) {
      MatrNumber m = new MatrNumber( l );
      return m;
   }
   public static MatrNumber fromString( String s ) {
      throw new UnsupportedOperationException( "not implemented yet");
   }
   public static MatrNumber createUniqueNew() {
      throw new UnsupportedOperationException( "not implemented yet");
   }
}
```

MatrNumberException

```
package thkoeln.st2.domainprimitives;

public class MatrNumberException extends
RuntimeException {
   public MatrNumberException( String msg ) {
      super( msg );
   }
}
```

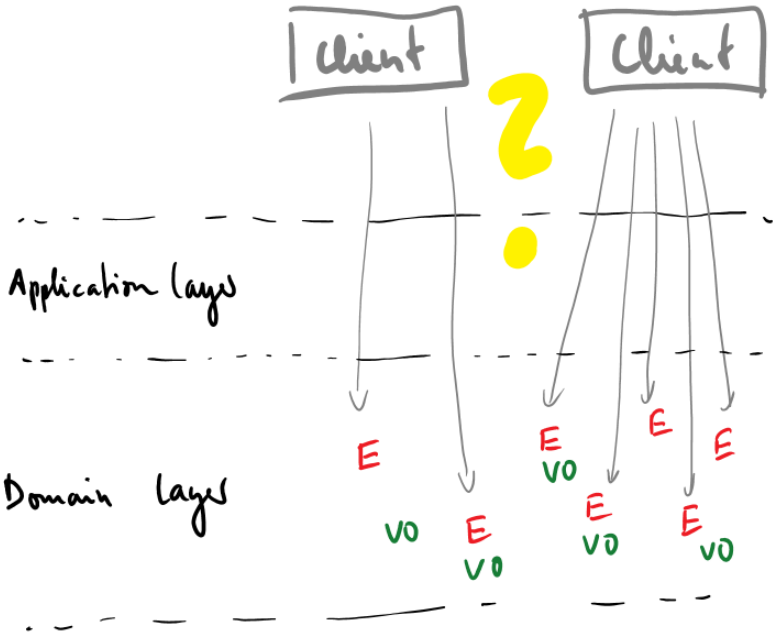
Was sind Aggregates?

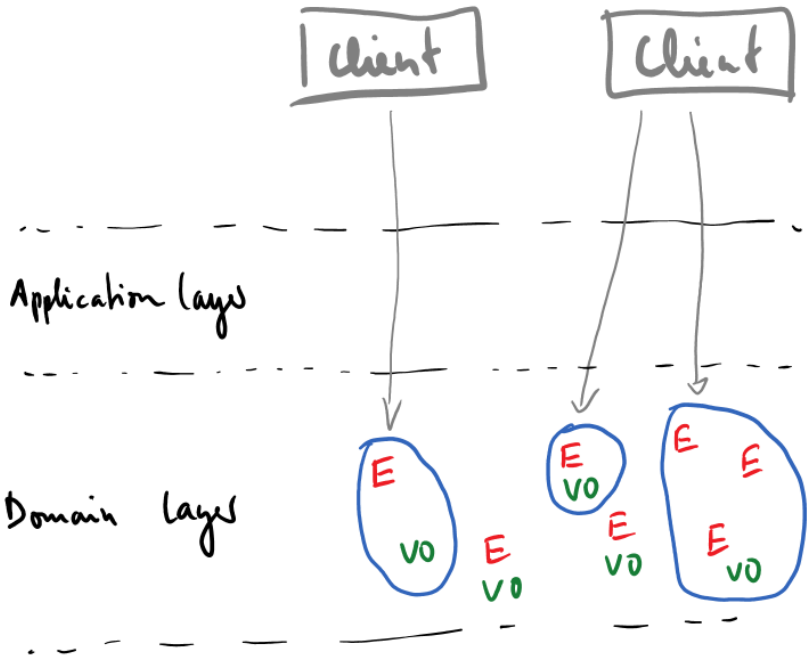


Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=KywRgZpLb5w

Big Picture: **Warum Aggregates wichtig sind**





- Entities, Value Objects und Repositories nutzen Sie, um Ihren Domain Layer – also die elementare Geschäftslogik im Backend – zu implementieren.
- Das alles machen Sie am Ende aber i.d.R. nur, um mit Clients via Schnittstellen (z.B. REST) auch darauf zugreifen zu können. Wenn Sie das machen – möchten Sie dann wahllos einfach ***ALLES*** nach außen sichtbar (und änderbar) machen (siehe Bild links)?
- Vermutlich nein. Und wenn Sie nur selektiv bestimmte Teile Ihrer Funktionalität über Schnittstellen nach außen sichtbar machen, welche Teile? Und welche Struktur haben dann die Schnittstellen? Aggregates (Bild rechts) sind bei dieser Frage eine sehr große Hilfe.

Definition eines Aggregate

- **Aggregate =**
  Geflecht von Entities mit fachlichem Zusammenhang

- Die Bestandteile des Aggregate haben Invarianten
  - Kann sich auf Attribute verschiedener inneren Entities beziehen
- Die **Wurzel-Entity** hat eine **globale** Identität
  - Identitäten der inneren Entities sind lokal zum Aggregate
- Nur diese Wurzel-Entity kann durch eine Query gesucht werden
  - keine Suche nach inneren Entities
- Referenzen von außen: nur auf die Wurzel-Entity
  - keine Referenzen auf innere Entities
- Lösch-Operationen löschen das gesamte Aggregate

Quelle: [Evans], S. 128 f.

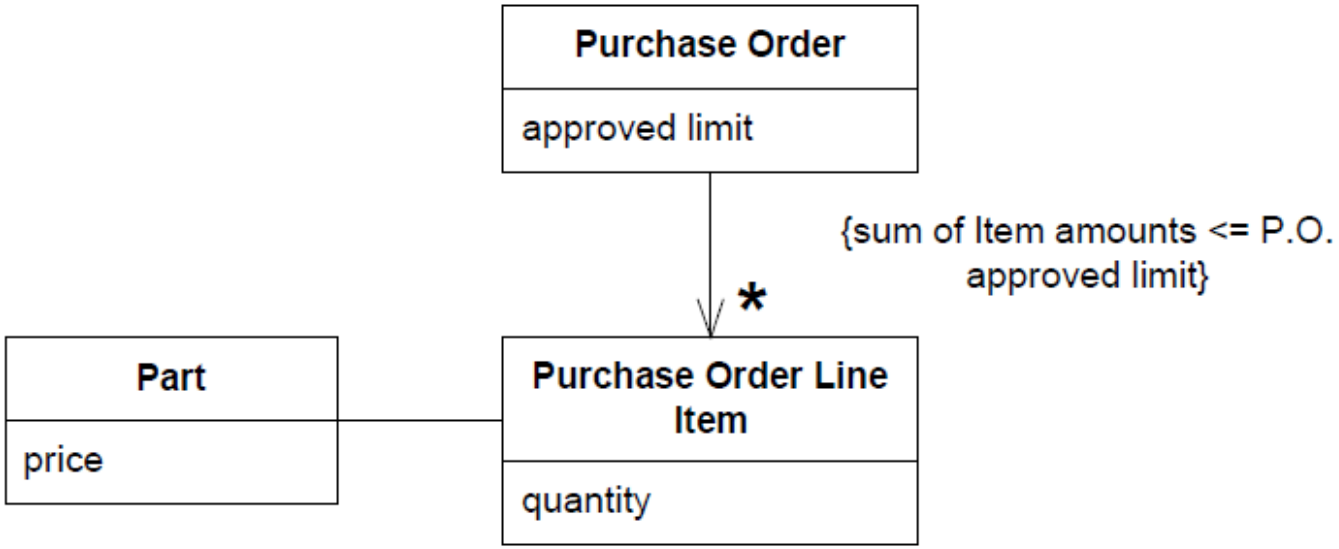
Definition Aggregate nach Evans

Now, to translate that conceptual AGGREGATE into the implementation, we need a set of rules to apply to all transactions:

- The root ENTITY has global identity, and is ultimately responsible for checking invariants.
- Root ENTITIES have global identity. ENTITIES inside the boundary have local identity, unique only within the AGGREGATE.
- Nothing outside the AGGREGATE boundary can hold a reference to anything inside, except to the root ENTITY. The root ENTITY can hand references to the internal ENTITIES to other objects, but those objects can only use them transiently, and may not hold onto the reference. The root may hand a copy of a value to another object, and it doesn't matter what happens to it, since it's just a value and no longer will have any association with the AGGREGATE.
- As a corollary to the above rule, only AGGREGATE roots can be obtained directly with database queries. All other objects must be found by traversal of associations.
- Objects within the AGGREGATE can hold references to other AGGREGATE roots.
- A delete operation must remove everything within the AGGREGATE boundary at once. (With garbage collection, this is easy. Since there are no outside references to anything but the root, delete the root and everything else will be collected.)
- When a change to any object within the AGGREGATE boundary is committed, all invariants of the whole AGGREGATE must be satisfied.

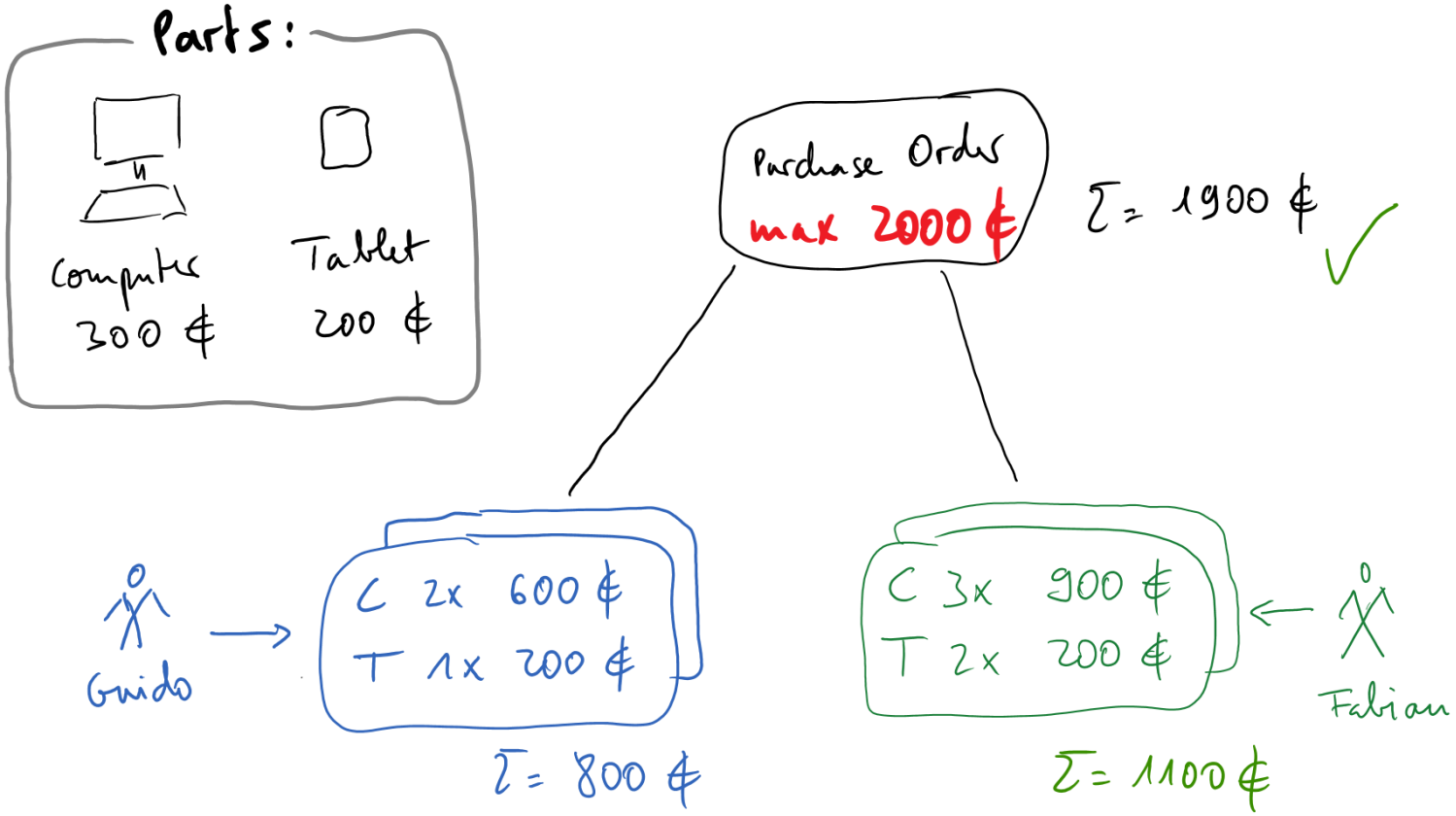
Quelle: [Evans], p. 92

Motivation für Aggregates am konkreten Beispiel (1)



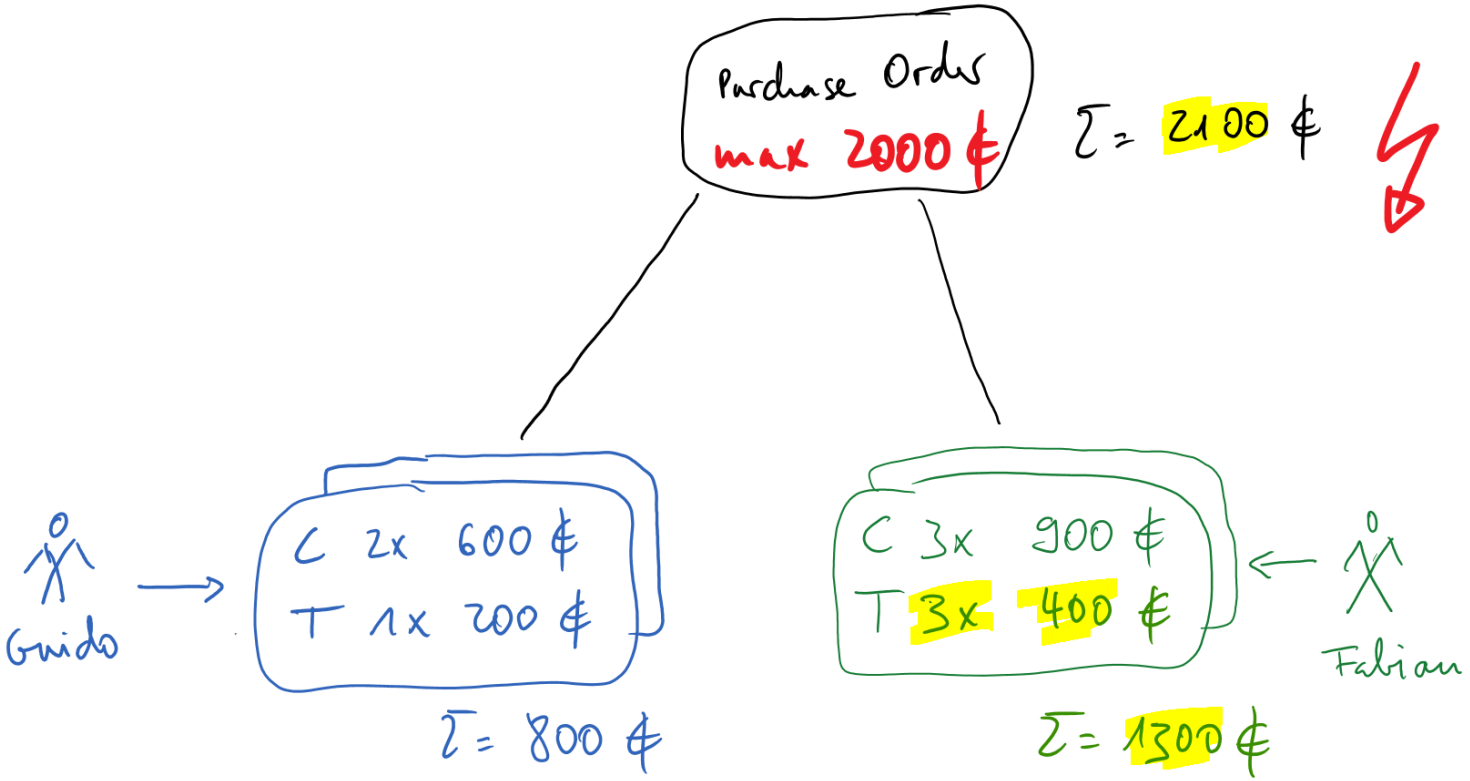
- (Quelle: [Evans], S. 93)
- Hier haben wir den Fall einer **Invariante** in der Beziehung zwischen Objekten.
- Eine *Purchase Order* besteht aus mehreren *Purchase Order Line Items*, die jeweils auf einen zu bestellenden *Part* verweisen.
- Dabei gibt es eine „Kreditlinie“: Die *Purchase Order Line Items* dürfen sich nicht zu mehr als dem *approved limit* aufsummieren.

Motivation für Aggregates am konkreten Beispiel (2)



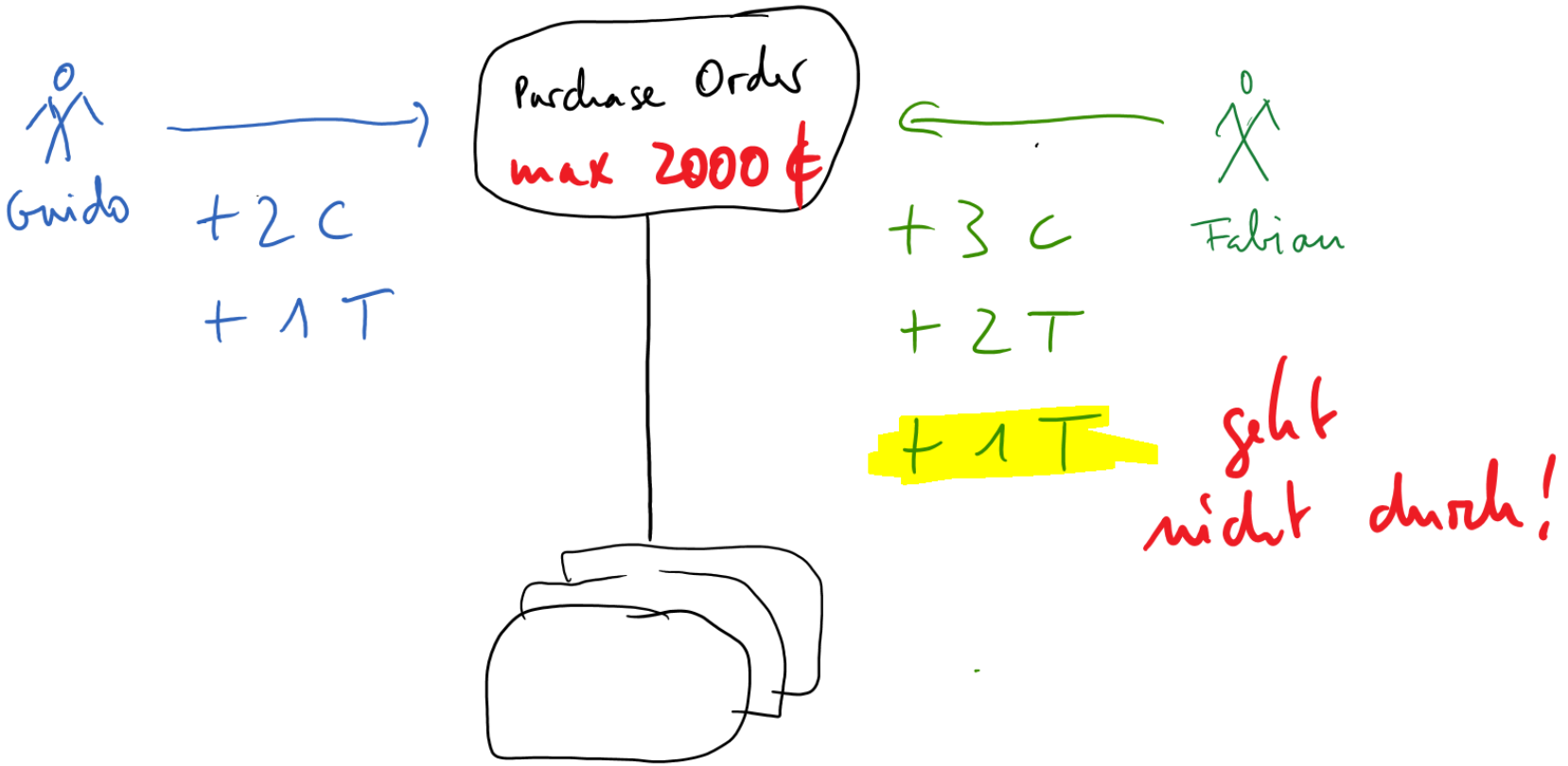
- Als konkretes Beispiel: Die wissenschaftlichen Mitarbeiter Guido und Fabian sollen neue Hardware für Mitarbeiter des Instituts anschaffen. Jeder von beiden ist für eine Arbeitsgruppe zuständig und sammelt da (unabhängig von einander) die Bestellungen ein. Es gibt Computer (C) und Tablets (T). Das Institut hat eine Kreditlinie von 2000€.
- Mit dem jetzigen Stand der Bestellungen ist die Kreditlinie eingehalten (1900€).

Motivation für Aggregates am konkreten Beispiel (3)



- Aber dann kommt in letzter Minute der Professor Gernot Gierig und will auch ein Tablet, weil seine Mitarbeiter auch eins haben. (Man weiß ja, wie diese Profs sind.) Fabian seufzt und ändert seine *Purchase Order Line Items* nochmal und nimmt das zusätzliche Tablet dort auf. Er macht seine Änderung lokal. Damit wird aber die Kreditlinie verletzt (Gesamtsumme jetzt 2100€).
- Wenn man sich jetzt Guido und Fabian nicht als Personen, sondern als Software-Clients vorstellt, wird das Problem klar: **Die Entity *Purchase Order Line Item* darf nicht unabhängig von *Purchase Order* verändert werden**. Sonst bekommt man schwer kontrollierbare Ausnahmesituationen.

Motivation für Aggregates am konkreten Beispiel (4)



- Besser: Hinzufügen von *Purchase Order Line Items* (oder generell: jeder Zugriff auf *Purchase Order Line Items*, auch für Änderungen) **läuft immer nur über die Klasse *Purchase Order***. Dort kann dann die Invariante geprüft werden.

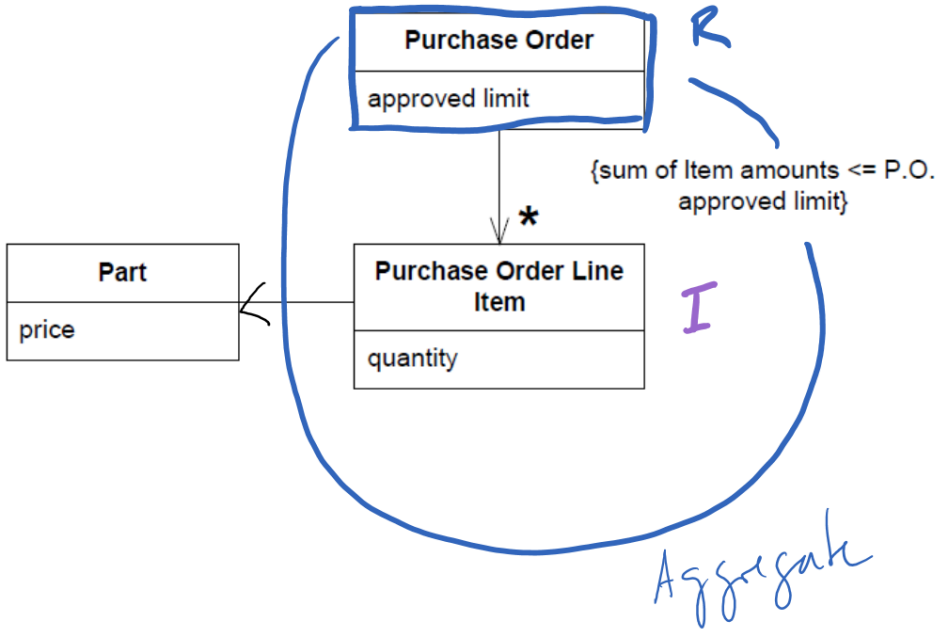
Checkliste für Aggregates

- Zwei Klassen mit gerichteter Beziehung **R** (root) ➔…➔ **I** (inner) **(*)** gehören zu einem Aggregate (mit **R** als Aggregate Root), wenn die ersten drei der nachfolgenden Bedingungen erfüllt sind:

☐	**I** tritt immer zusammen mit **R** auf. Es gibt keine Query nach **I**, außer zusammen mit einer Query nach **R**.
☐	Alle Zugriffe auf **I** erfolgen via **R**.
☐	Wenn **R** gelöscht wird, wird auch **I** gelöscht.
(☐)	Es kann Invarianten (**immer** gültige Bedingungen) in der Beziehung zwischen **R** und **I** (und ggfs. weiteren Klassen) geben. *(Optionale Bedingung)*

(*) Die Schreibweise **R** ➔…➔ **I** soll ausdrücken, dass es noch Zwischenklassen auf dem Pfad zwischen **R** und **I** geben kann. Die Beziehungsrichtung ist so, dass man von **R** zu **I** navigieren kann, aber nicht umgekehrt.

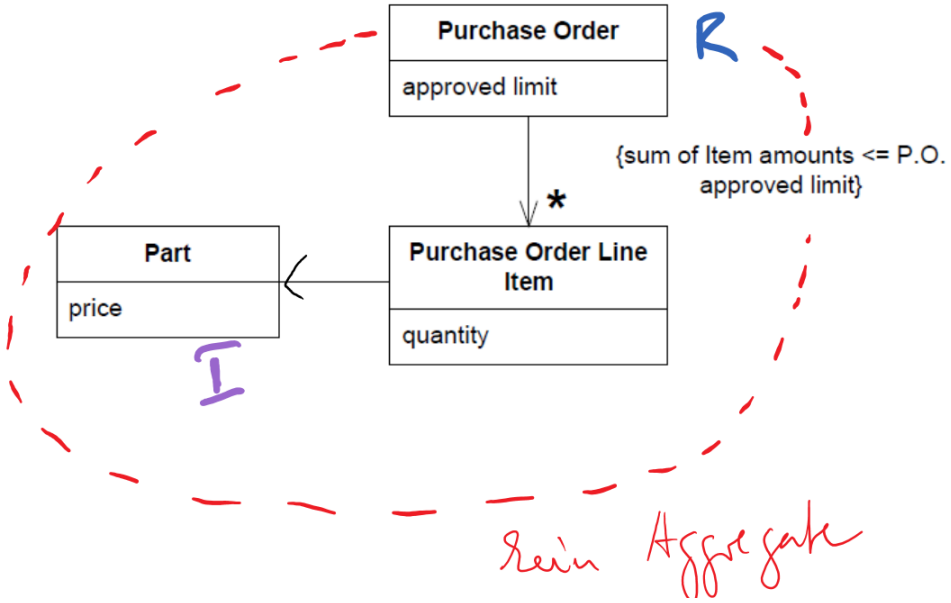
Test der Checkliste: *Purchase Order - POLI*



☑	**I** tritt immer zusammen mit **R** auf. Es gibt keine Query nach **I**, außer zusammen mit einer Query nach **R**.
☑	Alle Zugriffe auf **I** erfolgen via **R**.
☑	Wenn **R** gelöscht wird, wird auch **I** gelöscht.
(☑)	Es kann Invarianten (**immer** gültige Bedingungen) in der Beziehung zwischen **R** und **I** (und ggfs. weiteren Klassen) geben. *(Optionale Bedingung)*

- Testet man die Checkliste an der Beziehung zwischen *Purchase Order* und *Purchase Order Line Items*, stellt man schnell fest, dass die beiden zu einem Aggregate gehören (mit *Purchase Order* als Aggregate Root).

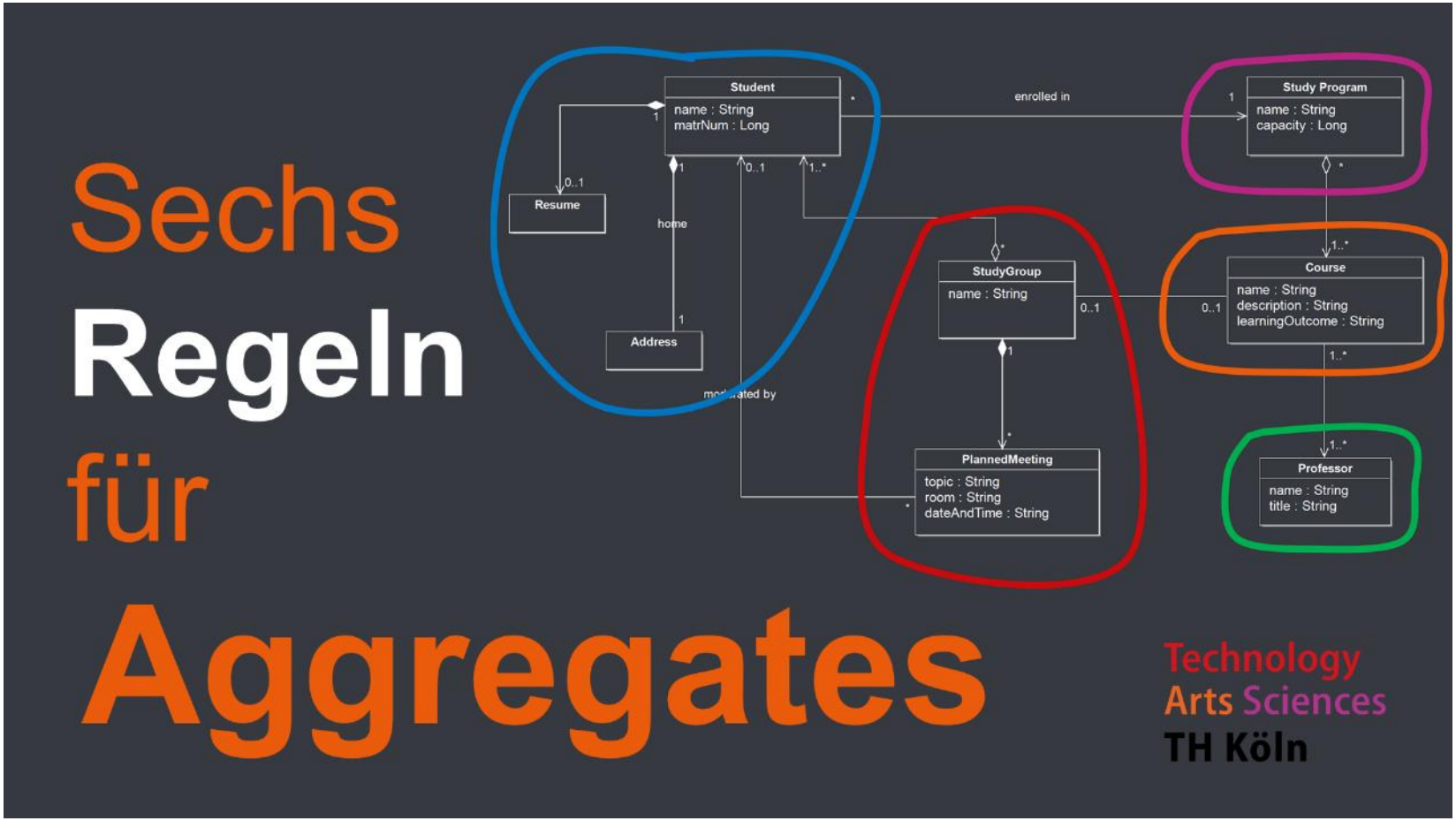
Test der Checkliste: *Purchase Order - Part*



☒	I tritt immer zusammen mit R auf. Es gibt keine Query nach I, außer zusammen mit einer Query nach R.
☒	Alle Zugriffe auf I erfolgen via R.
☒	Wenn R gelöscht wird, wird auch I gelöscht.
(☐?)	Es kann Invarianten (**immer** gültige Bedingungen) in der Beziehung zwischen R und I (und ggfs. weiteren Klassen) geben. *(Optionale Bedingung)*

- Dehnt man den Test auf *Purchase Order* und *Part* aus, dann ergibt sich ebenso schnell, dass die Bedingungen der Checkliste NICHT erfüllt sind – damit ist *Part* nicht Teil des Aggregates.

1. Man kann sich durchaus Queries nach *Part* vorstellen, die nichts mit einer *Purchase Order* zu tun haben (z.B. wenn man den Produktkatalog pflegt).
2. Ebenso möchte man Clients erlauben, Referenzen auf *Parts* außerhalb einer *Purchase Order* halten zu können, um z.B. den Preis anzupassen.
3. Eine kaskadierende Löschbeziehung ist ebenfalls absurd.

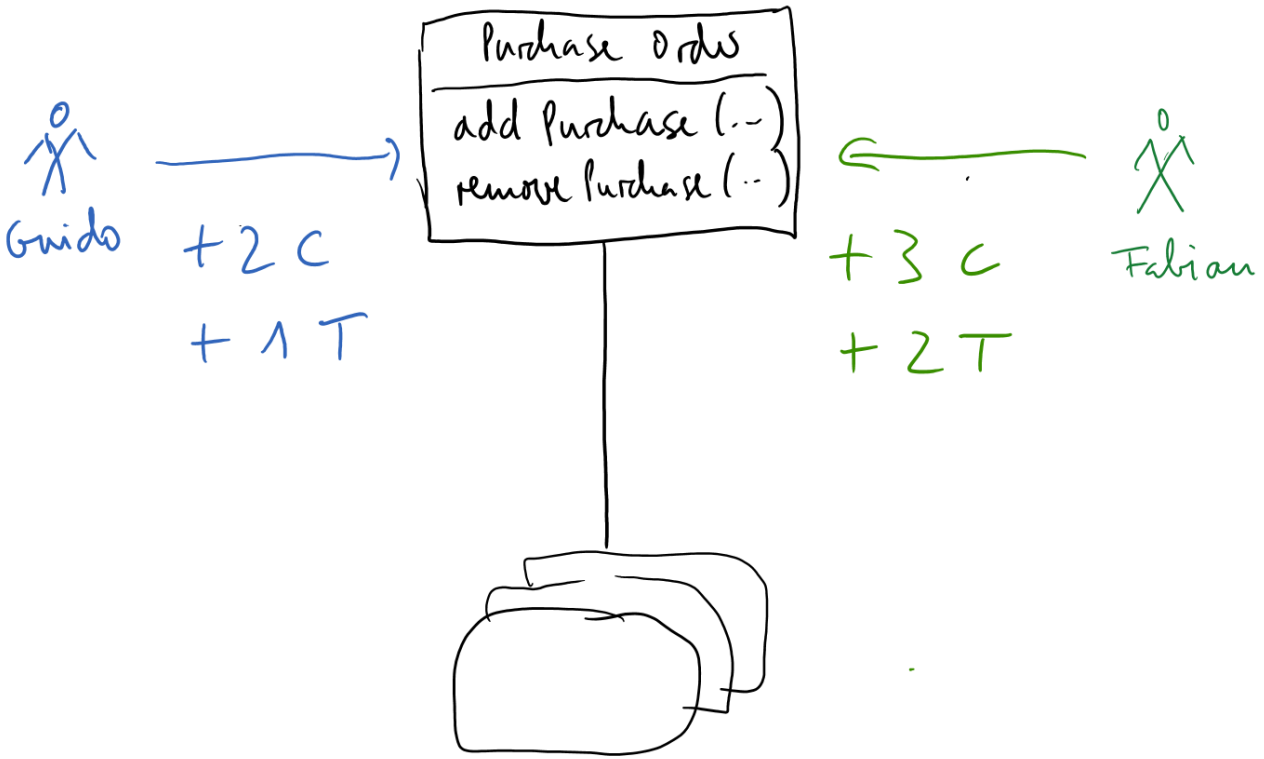


https://www.youtube.com/watch?v=WTau26feewU

Regel für Aggregates

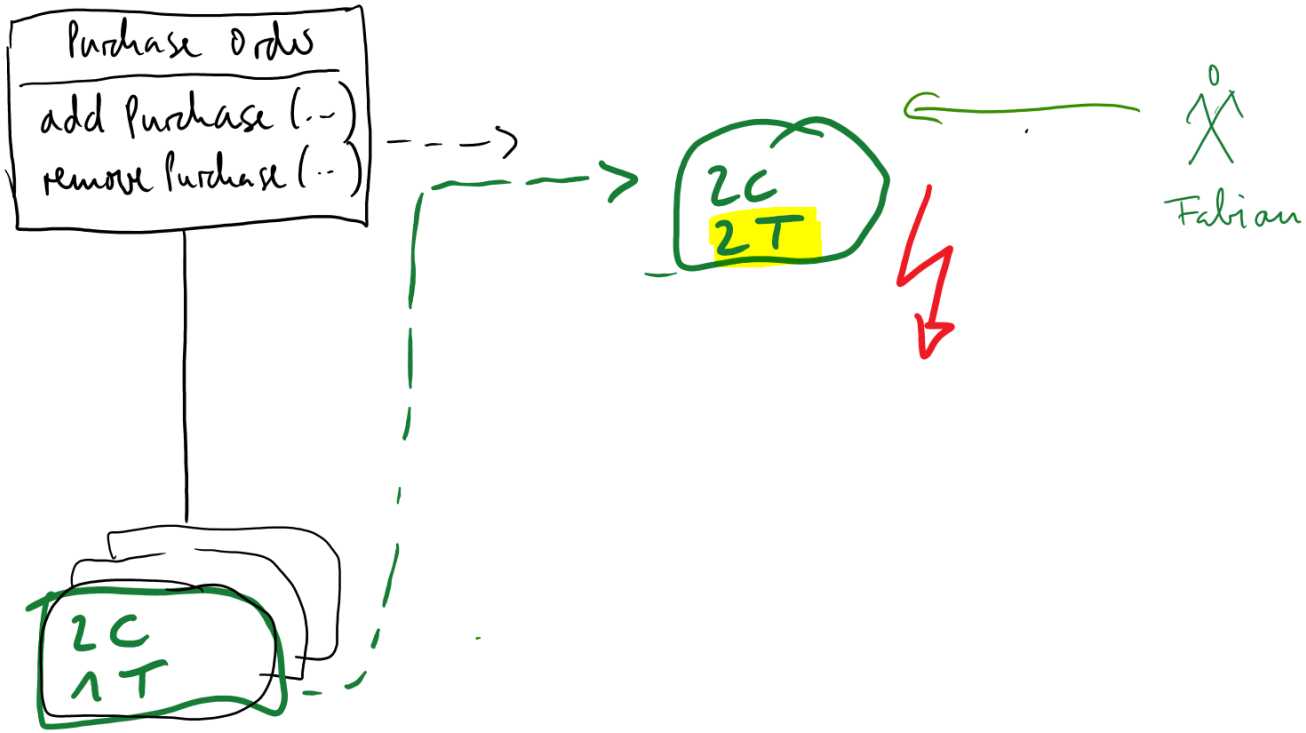
1. Referenzen von außen auf ein inneres Entity sind verboten.
   - Alle Zugriffe auf innere Entities müssen über Methoden des Aggregate Root erfolgen.
2. Aus dem Aggregate heraus darf es Referenzen auf andere Entities geben.
3. Entities können in höchstens einem Aggregat enthalten sein.
4. Value Objects können in mehreren Aggregaten enthalten sein.
5. Ein „alleinstehendes“ Entity ist immer auch ein Aggregate Root.
6. Beim Löschen des Aggregate Root werden alle inneren Entities auch gelöscht.

1. Referenzen von außen auf ein inneres Entity sind verboten. Alle Zugriffe auf innere Entities müssen über Methoden des Aggregate Root erfolgen.



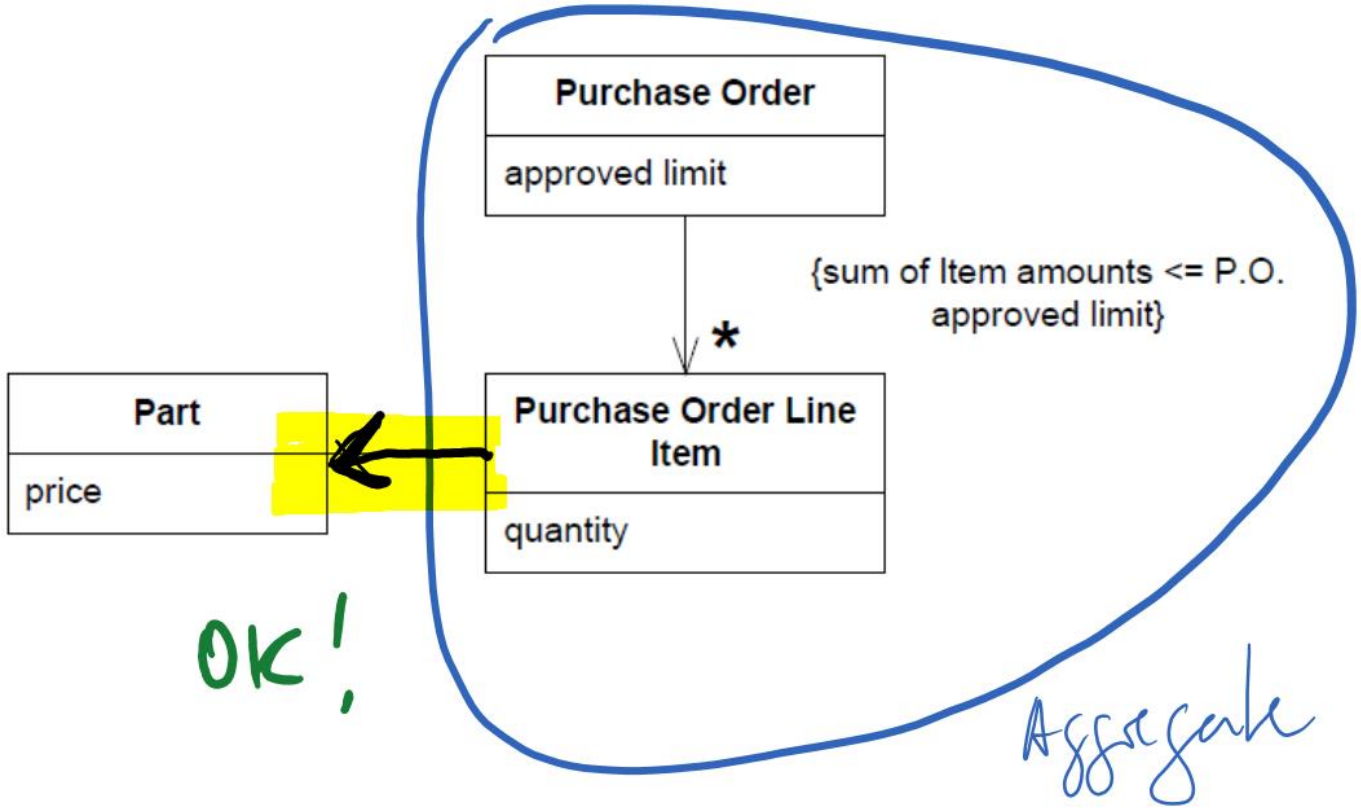
- Die Begründung liegt in der Definition des Aggregates. Siehe oben zu dem Beispiel von *Purchase Order* und *Purchase Order Line Items.*
- **=> Schauen wir uns gleich in dem Code-Beispiel noch einmal an.**

2. Wird ein inneres Entity von einer Methode zurückgegeben, so darf dies nur eine Kopie sein.



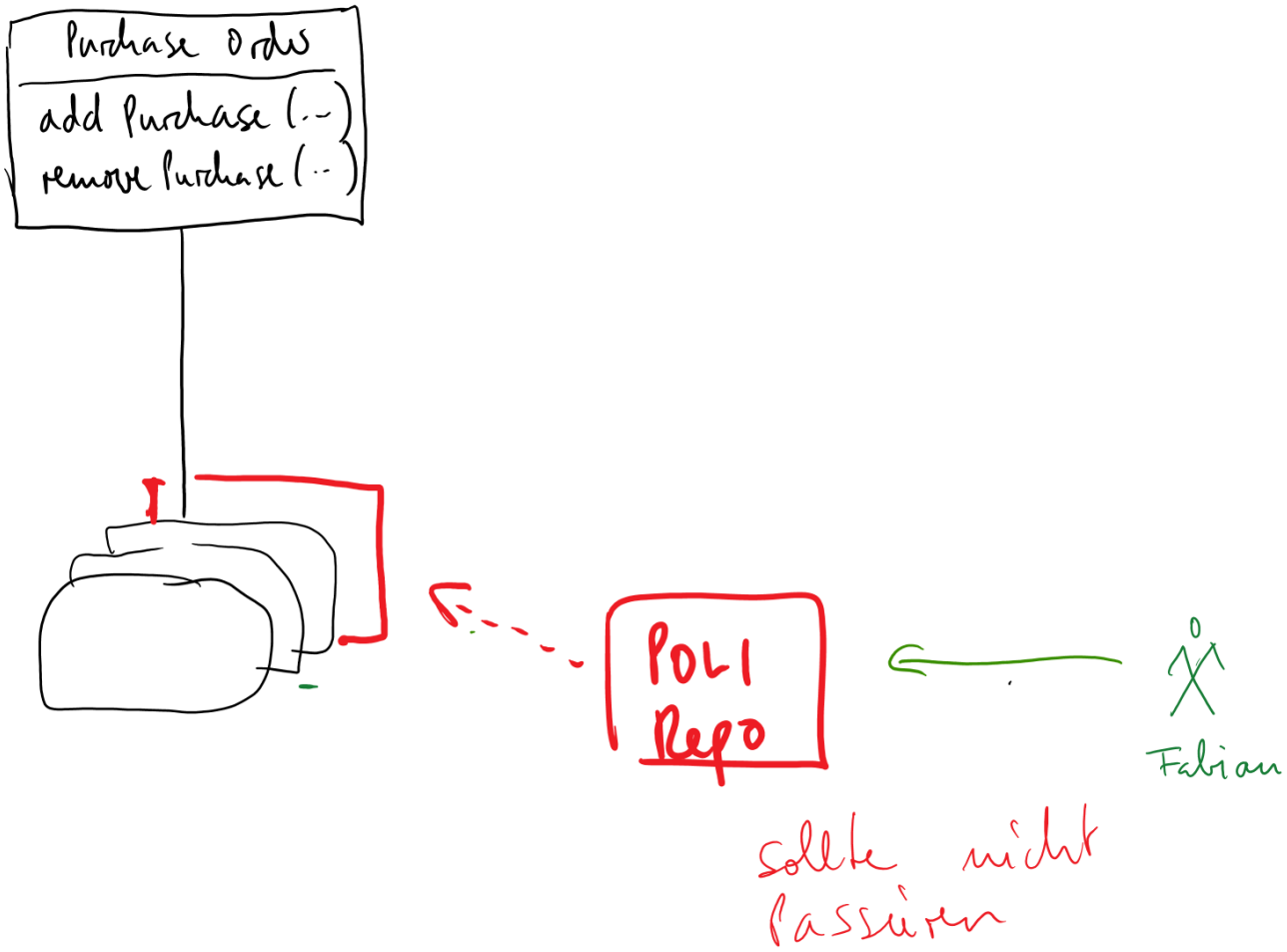
- Wird eine persistierbare Referenz (= ein Entity, das eine Session des ORM gebunden ist) zurückgegeben, dann könnte man sich eine Referenz auf inneres Entity geben lassen und diese dann verändern.
- Siehe oben zu dem Beispiel von *Purchase Order* und *Purchase Order Line Items.* Wenn man das XXX extern ändern könnte (und die Änderungen in der DB landen), tritt genau der Fall aus dem Beispiel ein, den man vermeiden möchte. Also darf man nur eine „Read-Only“-Kopie bekommen.
- **=> Schauen wir uns gleich in dem Code-Beispiel noch einmal an.**

3. Aus dem Aggregate heraus darf es Referenzen auf andere Entities geben.



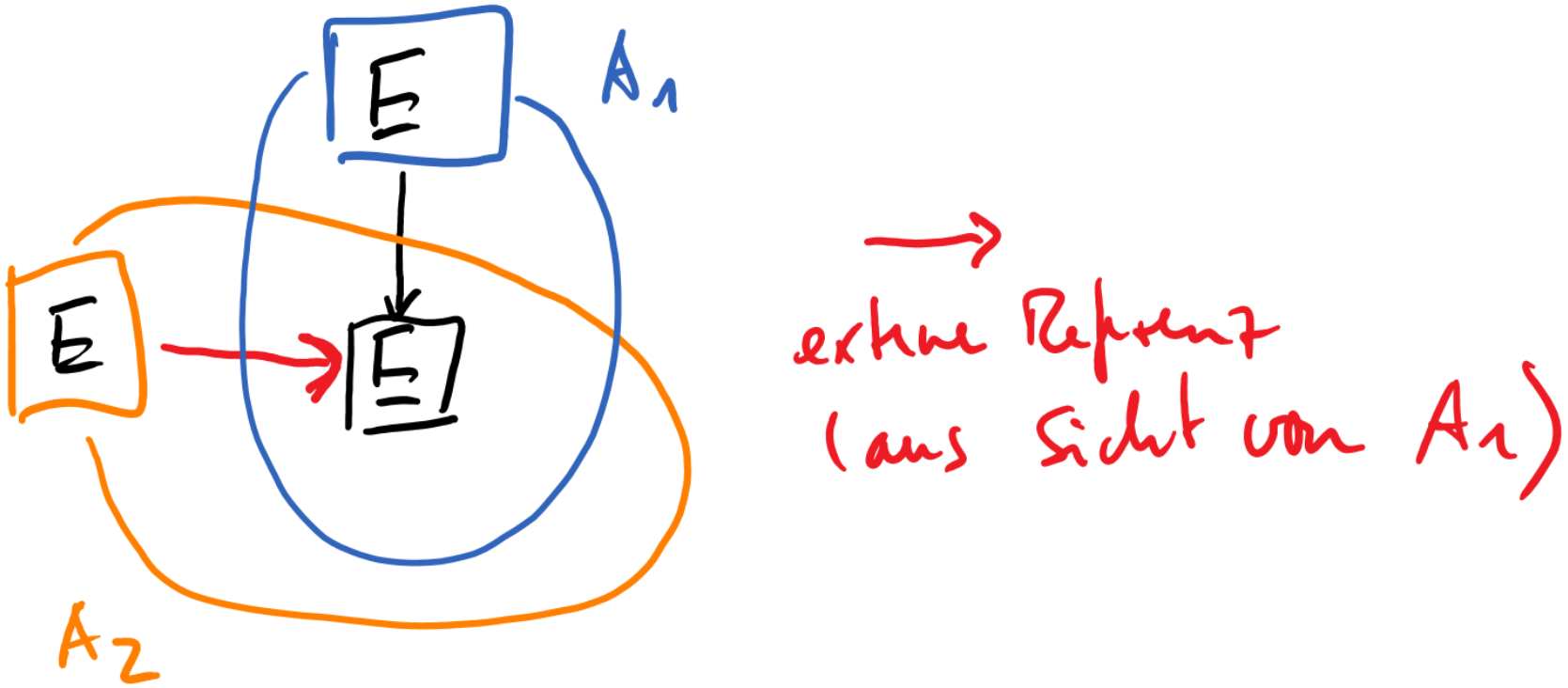
- Siehe *Purchase Order Line Items => Part*

4. Nur für Aggregate Roots gibt es Repositories.



- Wenn es für innere Entities Repositories gäbe, dann gäbe es eine Möglichkeit, solche inneren Entities unabhängig vom Repository zu erzeugen. Das aber ist verboten.

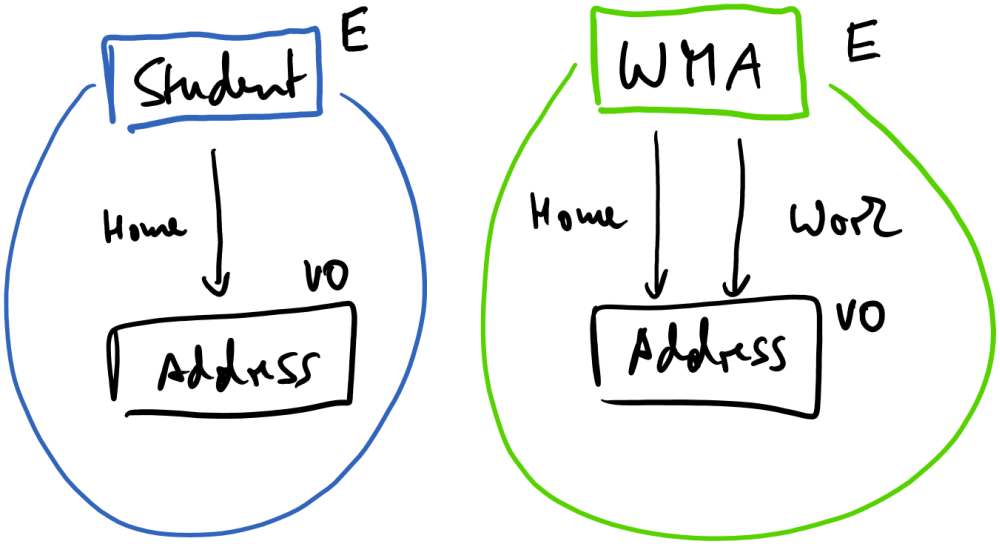
5. Entities können in höchstens einem Aggregat enthalten sein.



- Sonst gäbe es eine externe Referenz auf ein inneres Entity.

6. Value Objects können in mehreren Aggregaten enthalten sein.

7. Ein „alleinstehendes“ Entity ist immer auch ein Aggregate Root.



- Dies ist im Sinne der Idee von „Domain Primitives“, die als „Bibliothek“ Daten & Verhalten von Basis-Domänenobjekten abbilden. Diese sollten man zwischen Entities sharen können. *Beispiel:* Address sollte in unserem Student-Beispiel in verschiedenen Entities zur Verfügung stehen, z.B. auch *WMA* für Home- und Work-Address.
- Ein alleinstehendes Entity (hier *Student* und *WMA,* wenn man die Value Objects außer Acht lässt) ist sozusagen ein „Aggregate mit nur einem Element“ (und gleichzeitig dessen Root).
- Das macht uns das Formulieren von Regeln leichter, wenn wir uns mit REST-Interfaces und dem Exposing von Domänen-Objekten beschäftigen.

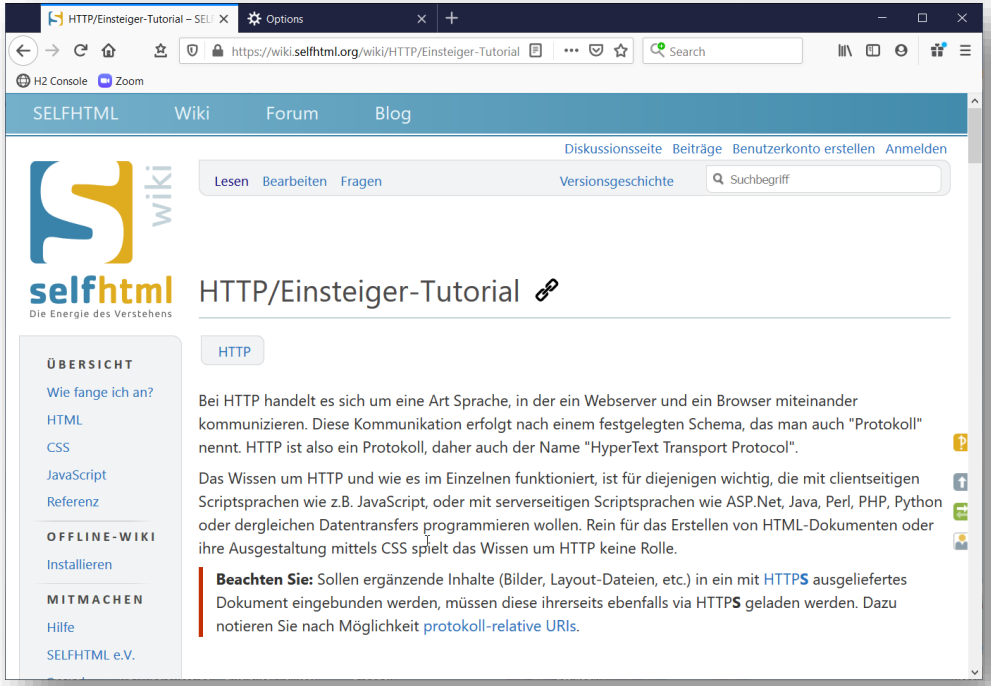
HTTP

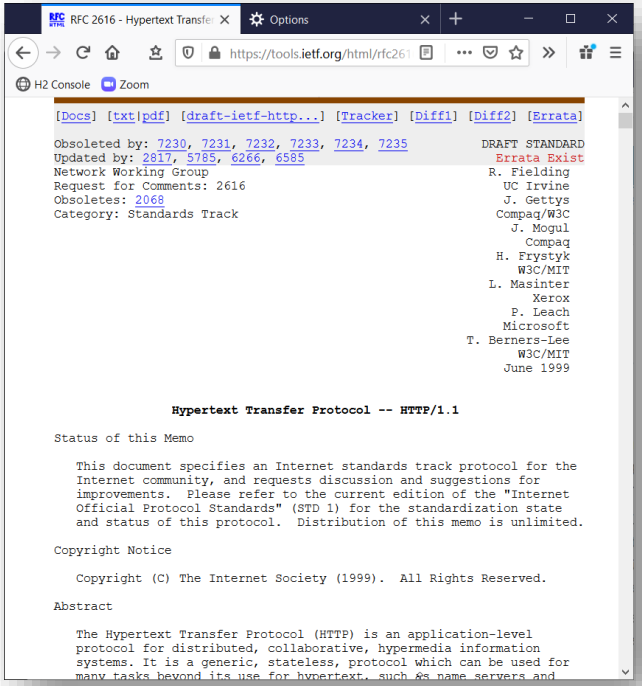
Grundlagen und Tools

Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=lVKuzvPjj40

HTTP Basics



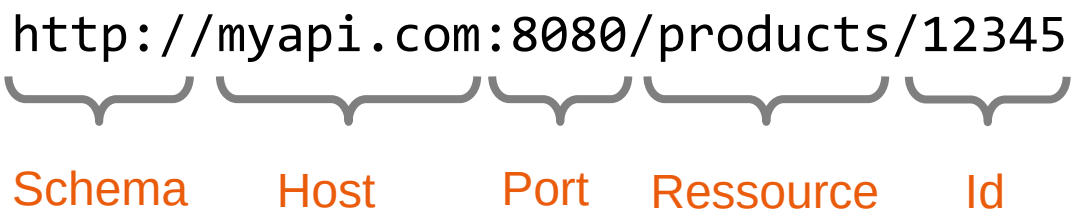


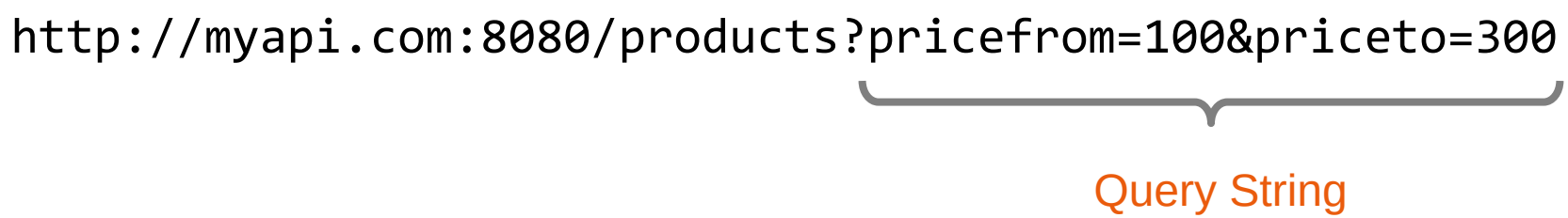
https://wiki.selfhtml.org/wiki/HTTP/Einsteiger-Tutorial

https://tools.ietf.org/html/rfc2616

- SelfHTML (links) ist schon seit Jahrzehnten die beste HTTP- und HTML-Source im deutschsprachigen Raum.
- Die originale RFC 2616 für HTTP 1.1 sieht herrlich old school aus (mit Tim Berners-Lee als einem der Autoren!!!).
  - Es ist aber tatsächlich eine wirklich gute Quelle, um manche Sachen mal im Original nachzugucken – z.B. ist 409 wirklich der richtige Return Code, wenn man einen Zeitschlitz für einen Arztbesuch buchen will, der leider gerade eben schon vergeben wurde?
  - **UPDATE**: Es gibt ein aktuellere Versionen, z.B: https://datatracker.ietf.org/doc/html/rfc7231

Bestandteile einer URL





Request

Request Header

```
PATCH /public-api/users/812 HTTP/1.1
Host: gorest.co.in
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:55.0)
Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;
                  q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
```

Request Body

```
{
    “name”: “Stefan”
}
```

Header-Beispiele auf dieser und der nächsten Seite nach https://wiki.selfhtml.org/wiki/HTTP/Einsteiger-Tutorial

Response

Response Header

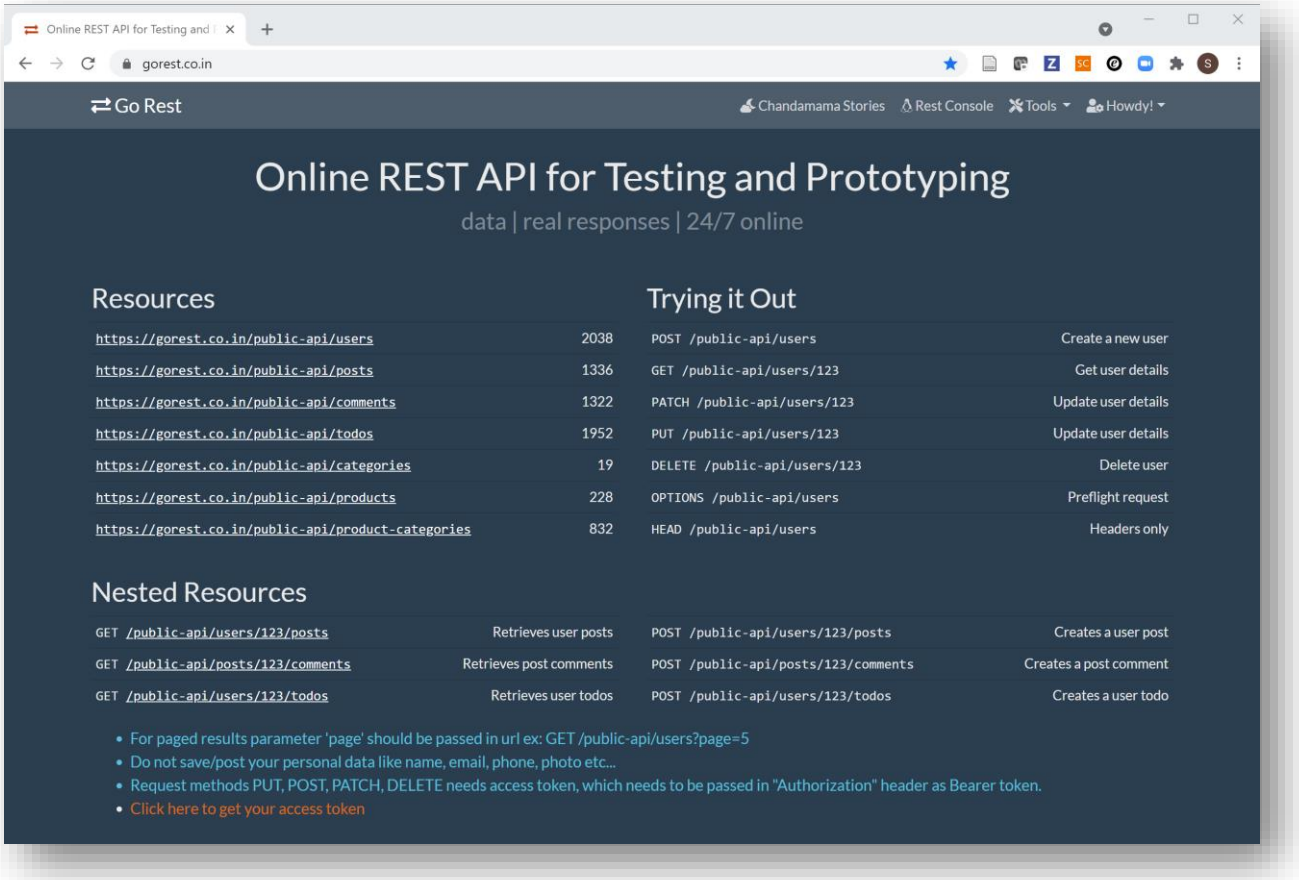
```
HTTP/1.1 200 OK
Date: Mon, 28 Aug 2017 09:02:55 GMT
Content-language: de-formal
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Last-Modified: Tue, 22 Aug 2017 08:15:31 GMT
Content-Encoding: gzip
Content-Length: 7986
Content-Type: text/html; charset=UTF-8
```

Response Body

```
{
   "id": 8,
   "name": "Stefan",
   "email": x@y.de
}
```

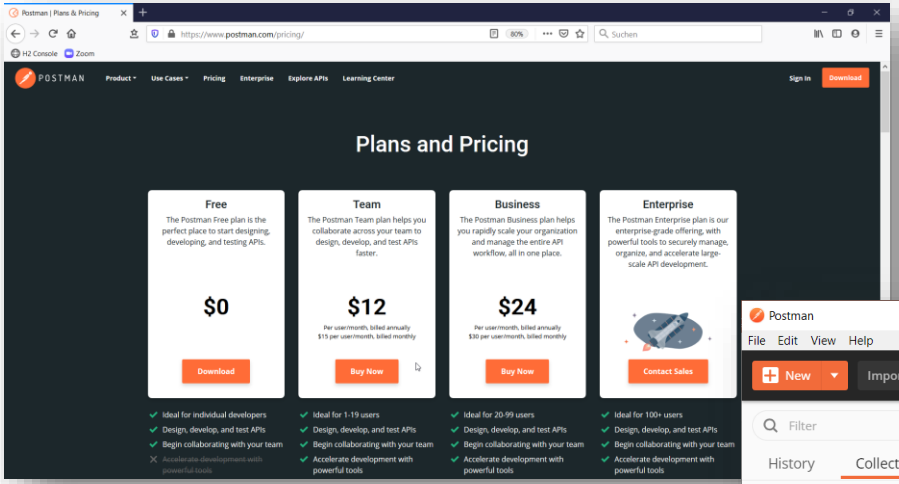
Öffentliches REST-API zum Ausprobieren (Fake-Daten)

https://gorest.co.in/



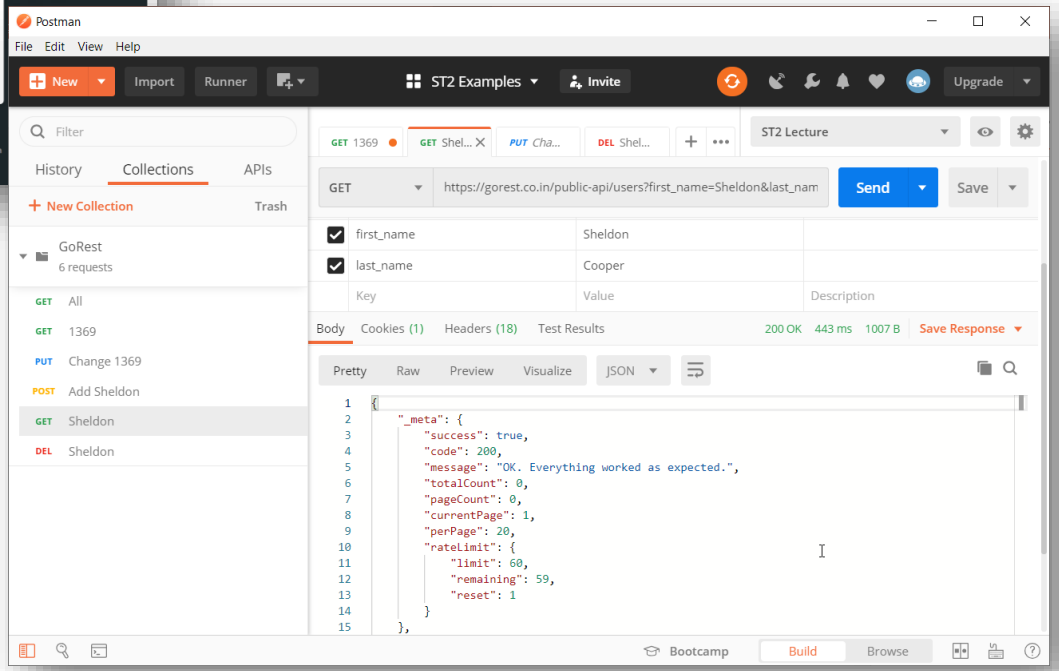
- Man muss sich (leider) einloggen, um das API sinnvoll nutzen zu können.
- Nach dem Log-In (z.B. mit Google-Account) kann man sich einen API Key generieren, den man dann etwa in Postman verwenden kann.
- Über die Console kann man die REST Calls auch direkt auf der Webseite ausführen.

Postman



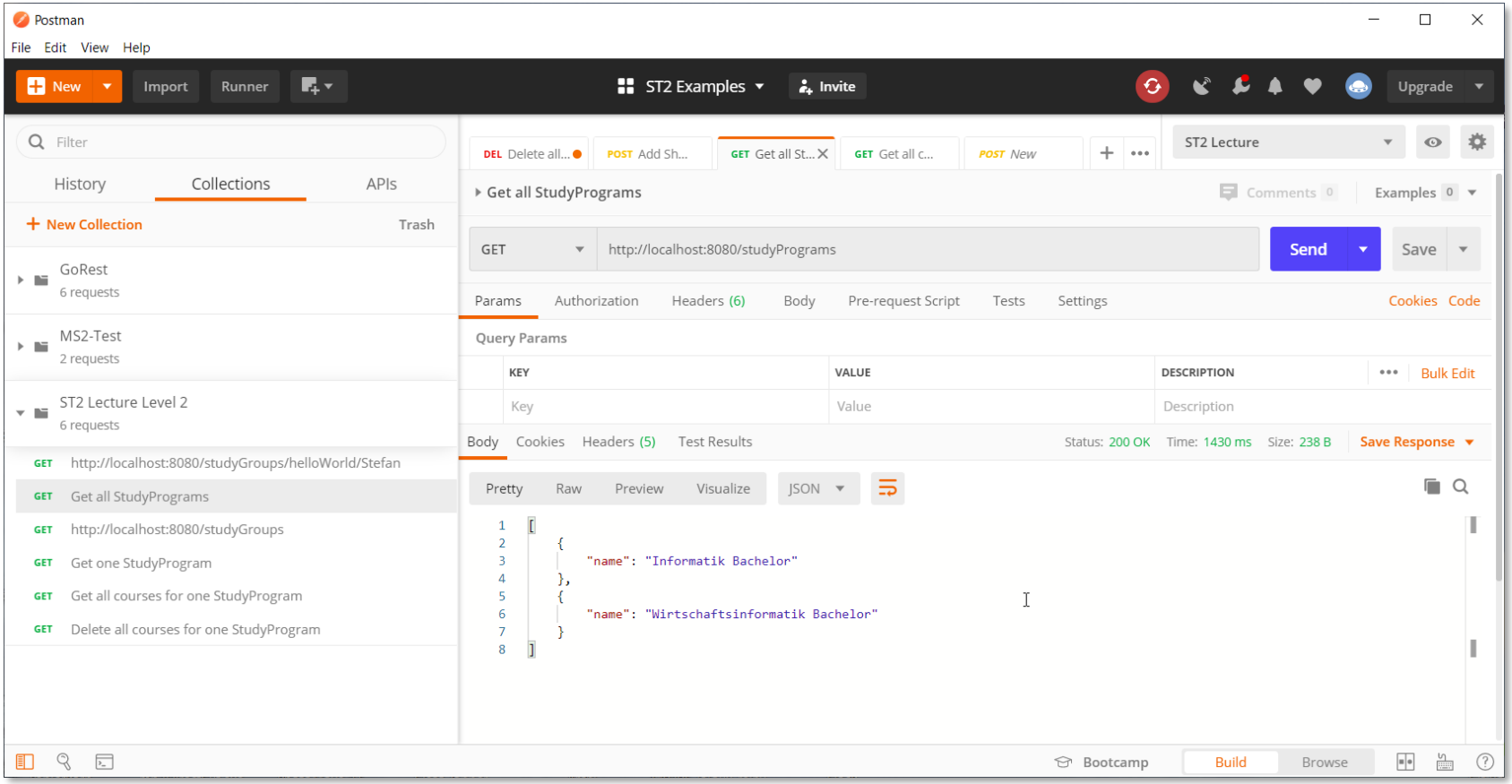
1. Anmelden für „Free Plan“
2. Herunterladen und Installieren (win, mac, linux)

https://www.postman.com/



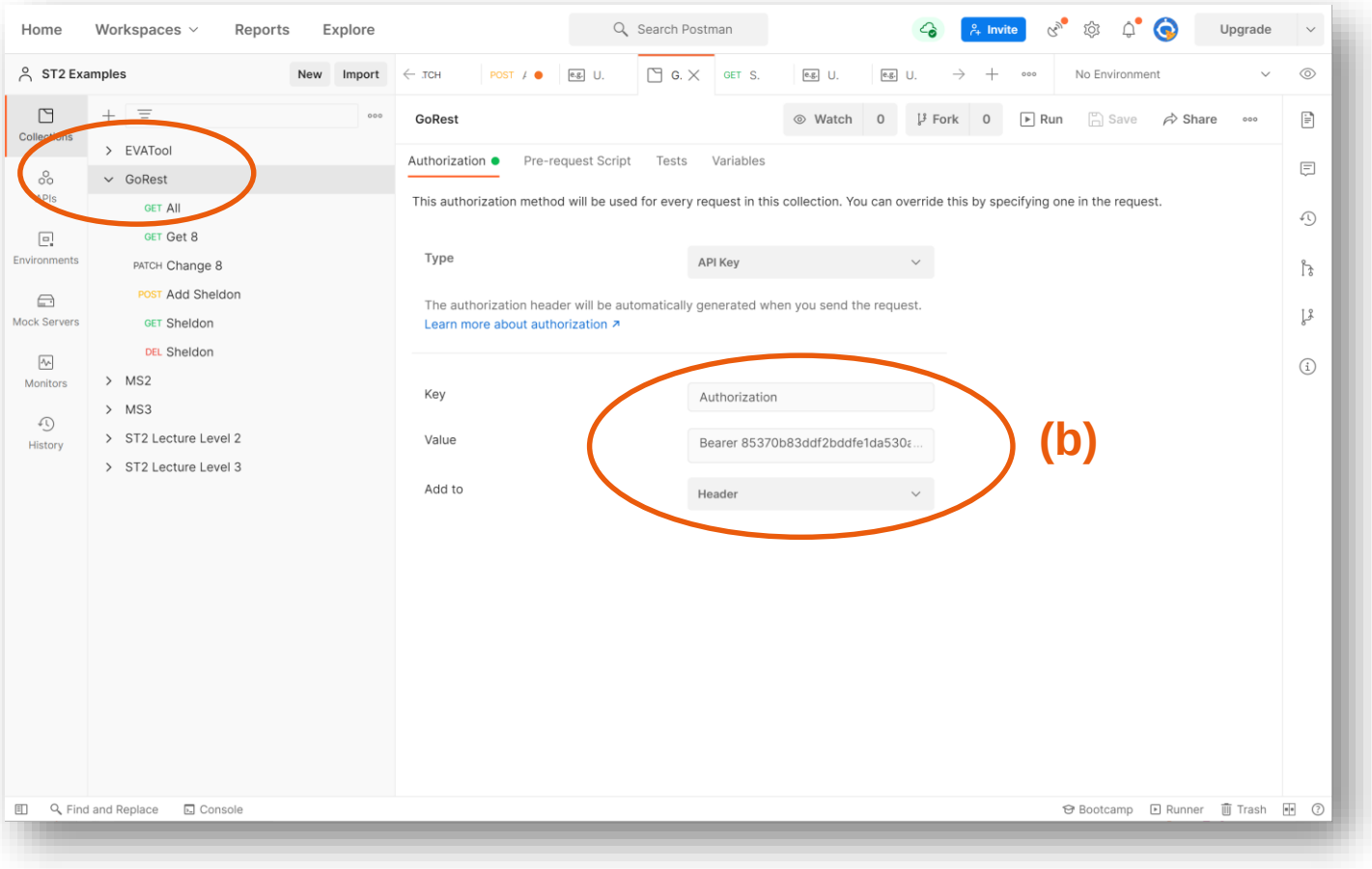
- Postman ist ein sehr praktisches Tool, wenn man selbst APIs entwickelt.
- Man kann die Nutzung bis hin zu ganzen REST-Testsuites ausbauen.
- Seine große Stärke spielt es aber schon als handliches “Schweizer Taschenmesser” aus, mit dem man sein eigenes API abfragen, testen und ausprobieren kann.

Postman als Tool



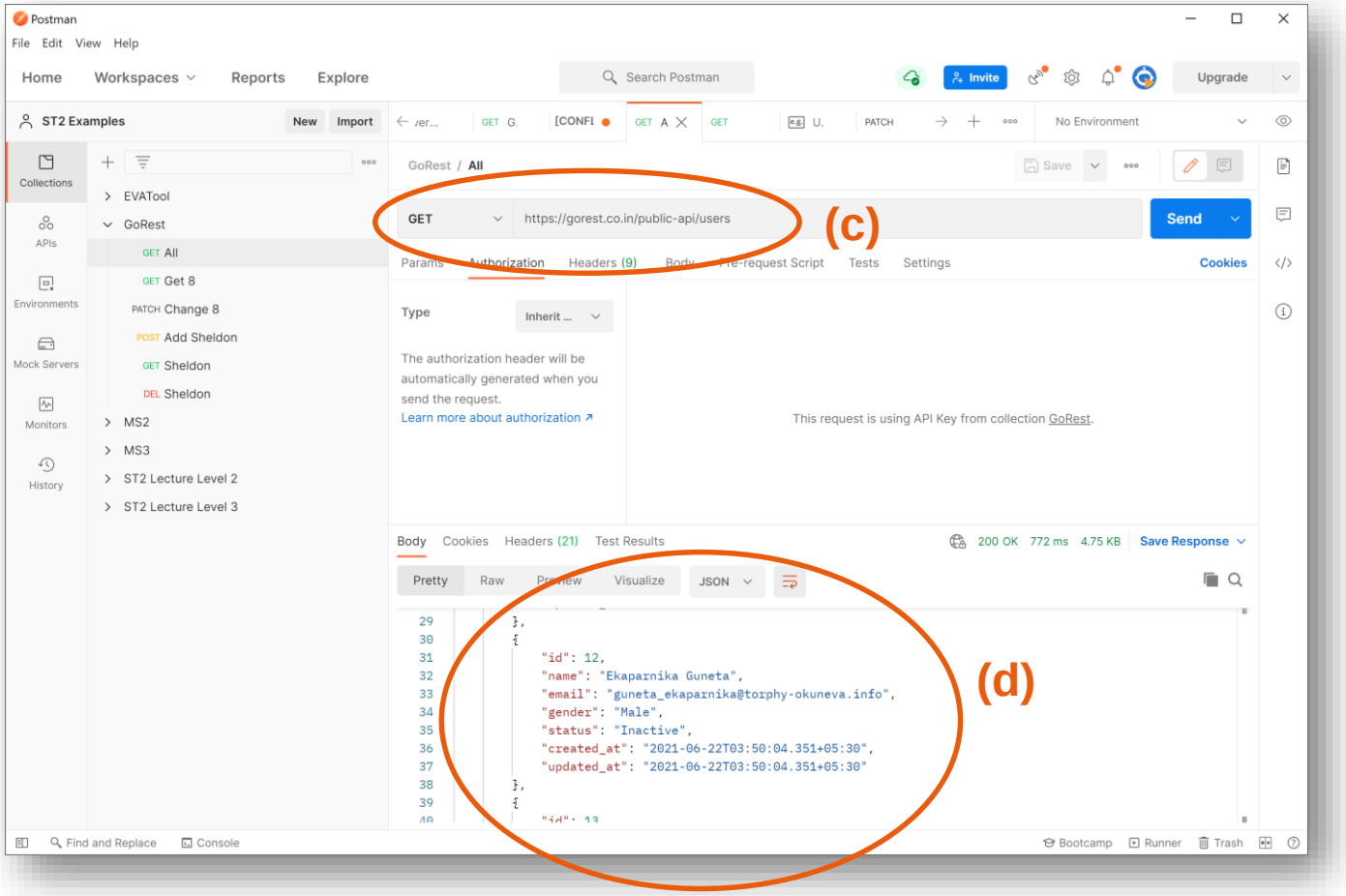
- Postman ist ein sehr gutes Tool, um REST-API-Entwicklung zu begleiten und zu testen.
  - Es gibt auch Alternativen, die Ähnliches können.
- Postman bietet sich an, um zu verstehen, was gerade passiert, als Alternative zum Debugging. Man kann auch ganze Test-Suites dort implementieren (in der Vorlesung nicht gemacht)

Ausprobieren von Postman mit `gorest.co.in` (1)



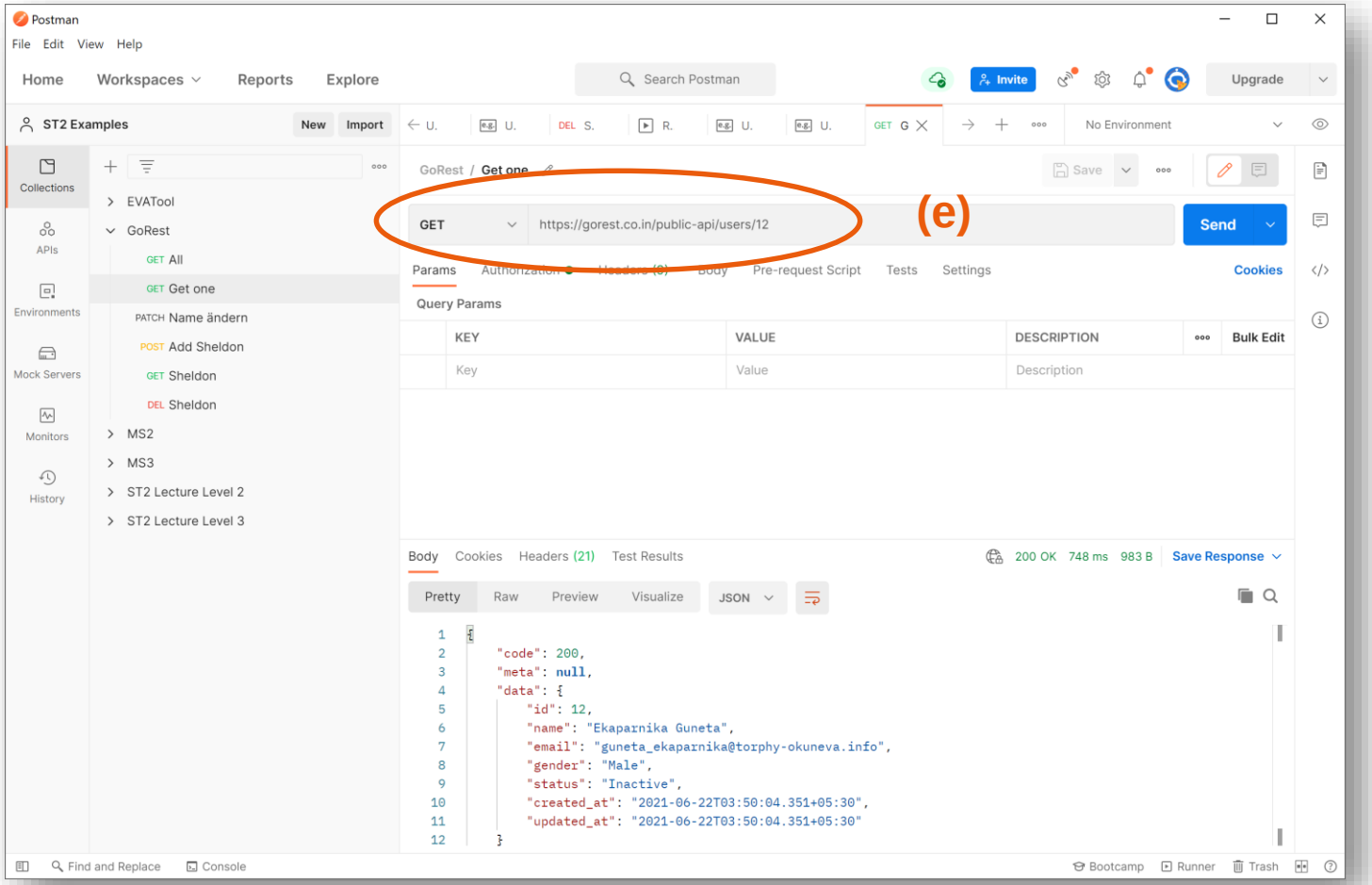
- Collection anlegen (a)
- Den API Key von gorest sollte man in der Collection ablegen (bitte als Ganzes kopieren, inklusive des “Bearer” am Anfang, sonst geht’s nicht) (b)
  - Dann können die einzelnen Aufrufe den API Key von Collection erben.

Ausprobieren von Postman mit `gorest.co.in` (2)



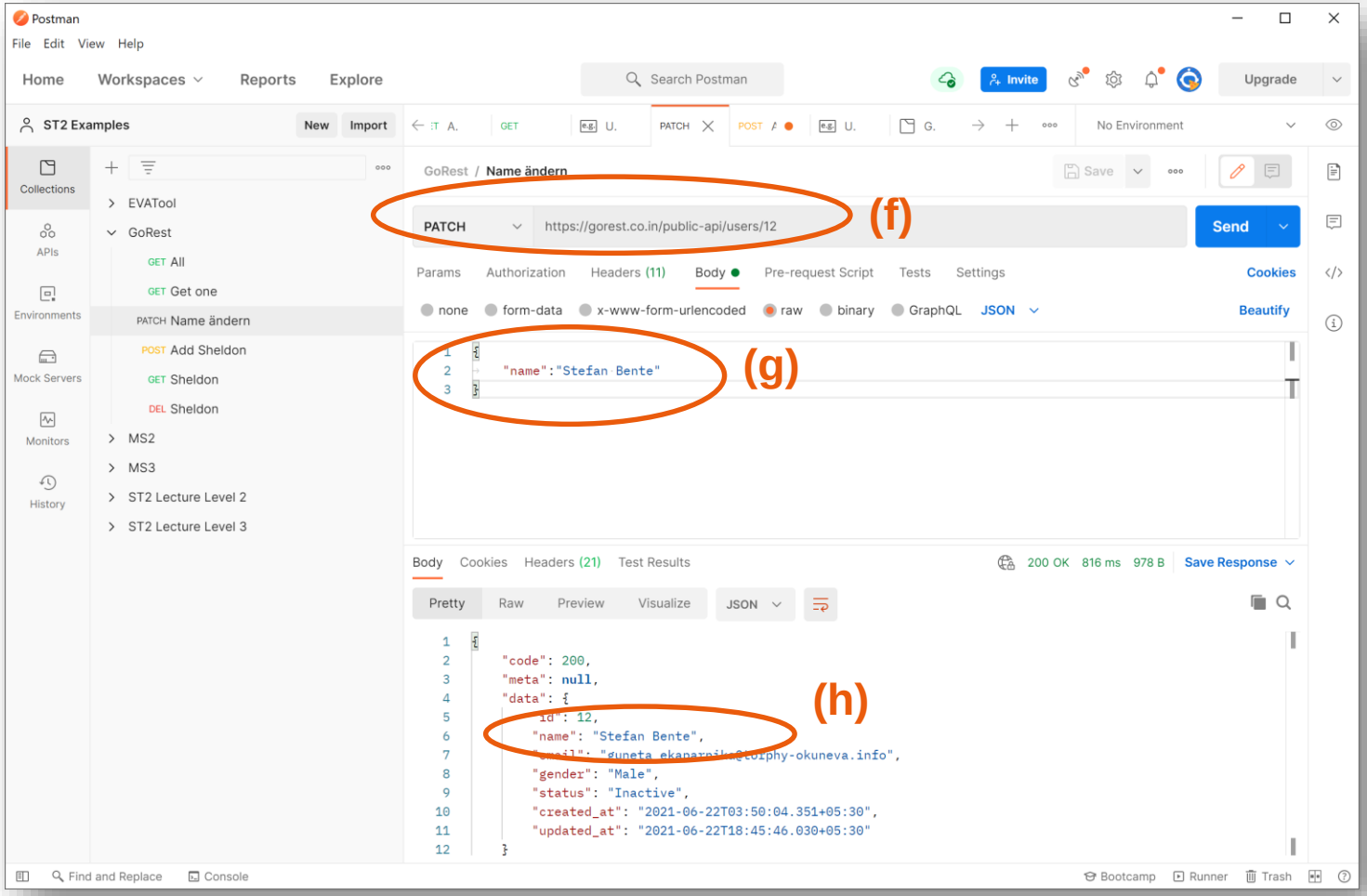
- URI angeben – hier ein “GET all” auf /users (c)
- Das Ergebnis kommt im Request Body zurück. (d)

Ausprobieren von Postman mit `gorest.co.in` (3)



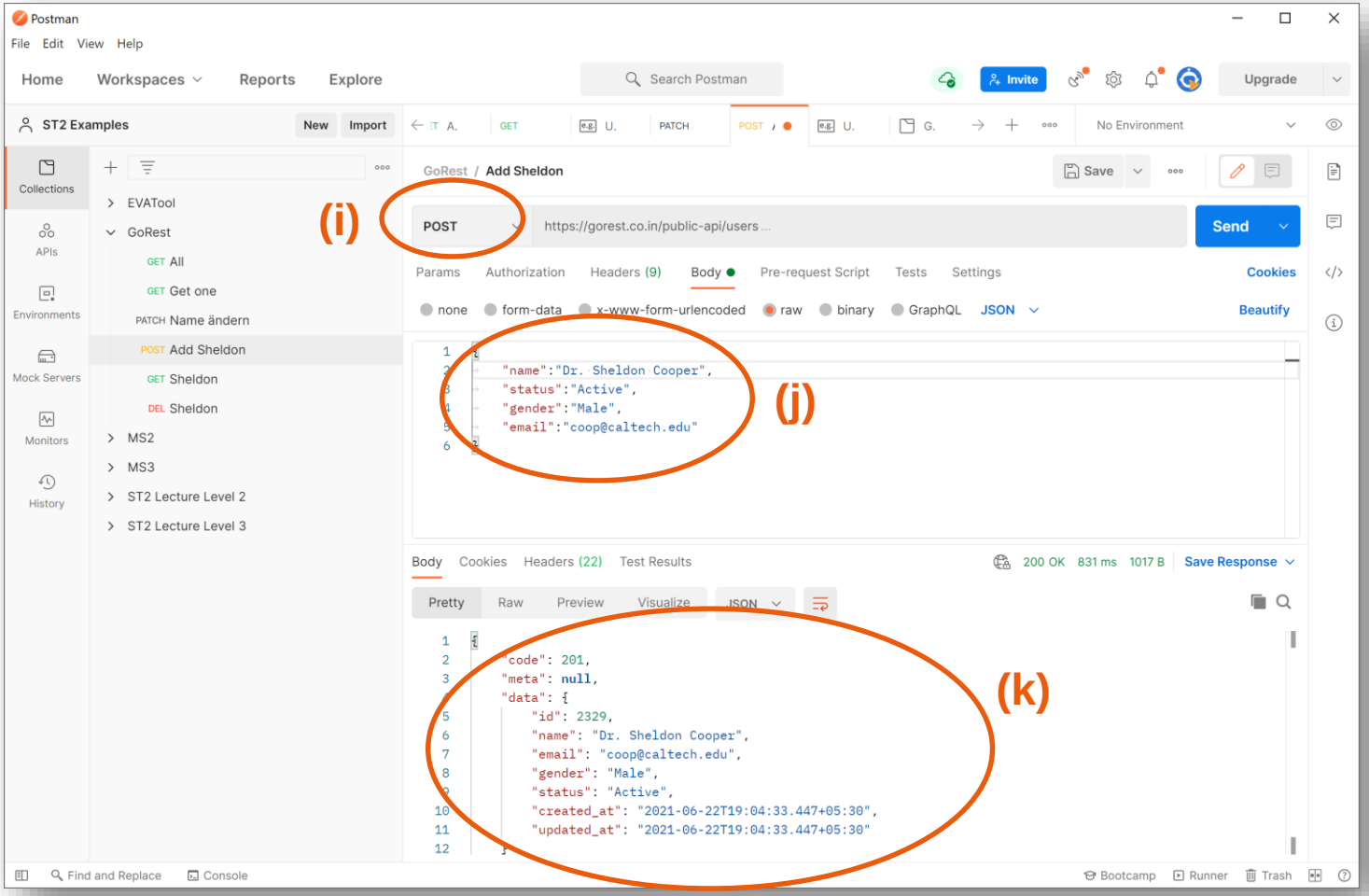
- Wenn man eine spezifische ID hinzufügt (hier die 12), dann hat man ein “GET one” (e)

Ausprobieren von Postman mit `gorest.co.in` (4)



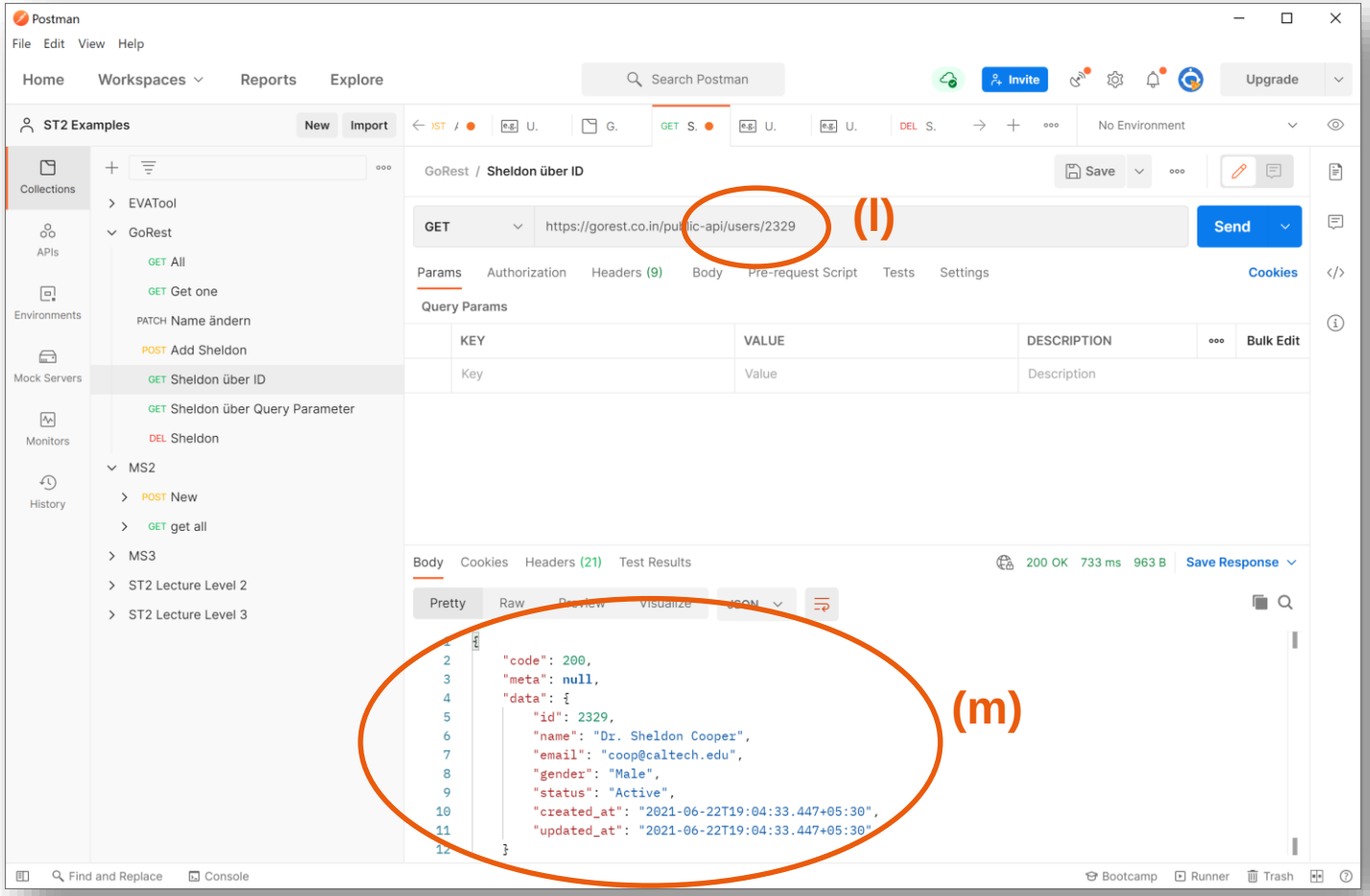
- Mit PATCH kann man einzelne Attribute einer Resource ändern … (f)
- Den / die neuen Attributwert(e) gibt man im Request Body an (g)
- Die geänderte Resource kommt im Request Body zurück (h) – wie man sieht, hat die Änderung geklappt.

Ausprobieren von Postman mit `gorest.co.in` (5)



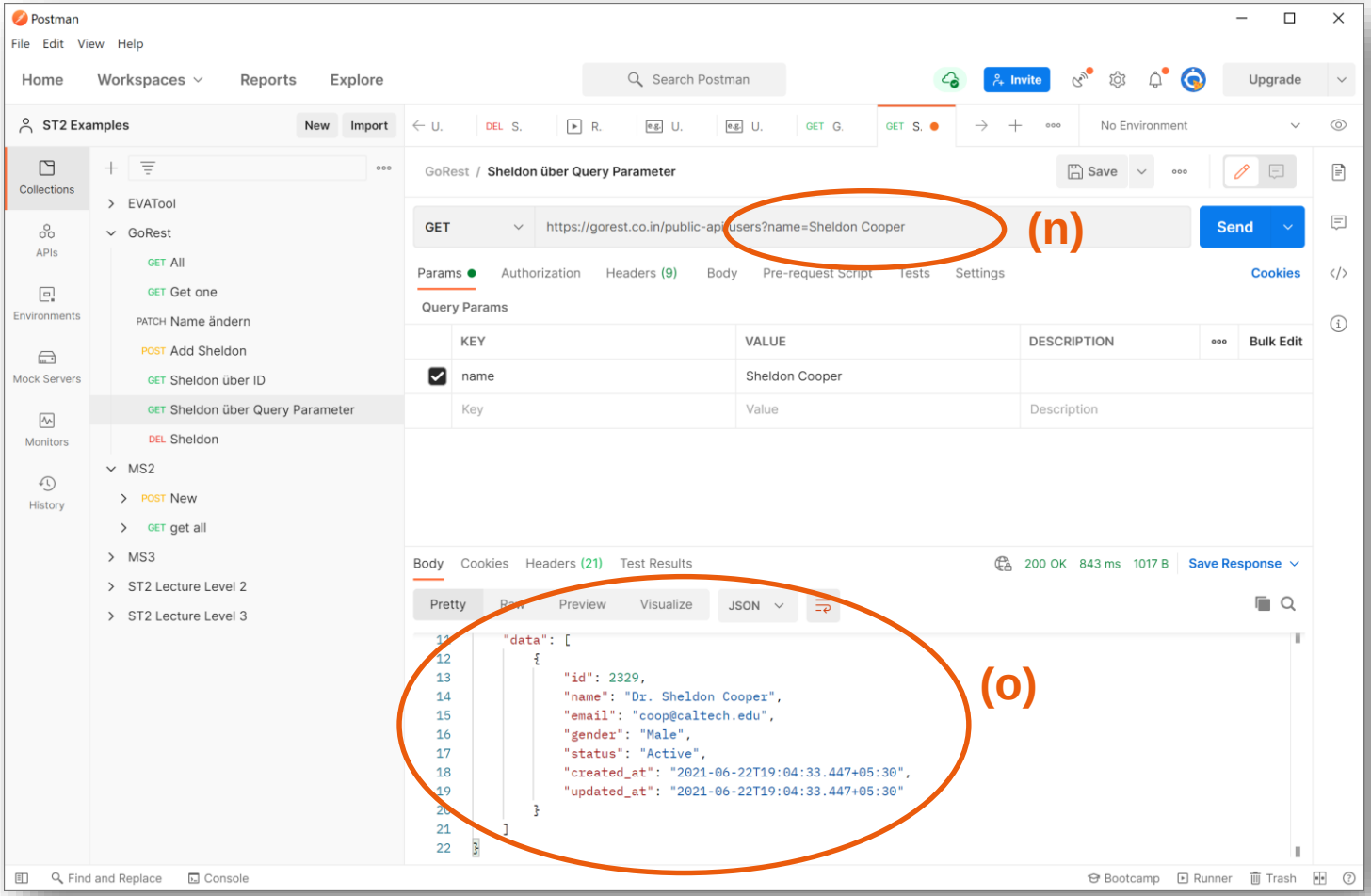
- Mit POST (i) kann man eine neue Resource anlegen, hier Sheldon Cooper. Die Daten dafür stehen im Request Body (j).
  - Eine ID fehlt natürlich noch – die wird vom Server vergeben.
- Den / die neuen Attributwert(e) gibt man im Request Body an (g)
- Die neue Resource kommt im Request Body zurück (k) – zusammen mit der neuen ID (hier: 2329).

Ausprobieren von Postman mit `gorest.co.in` (6)



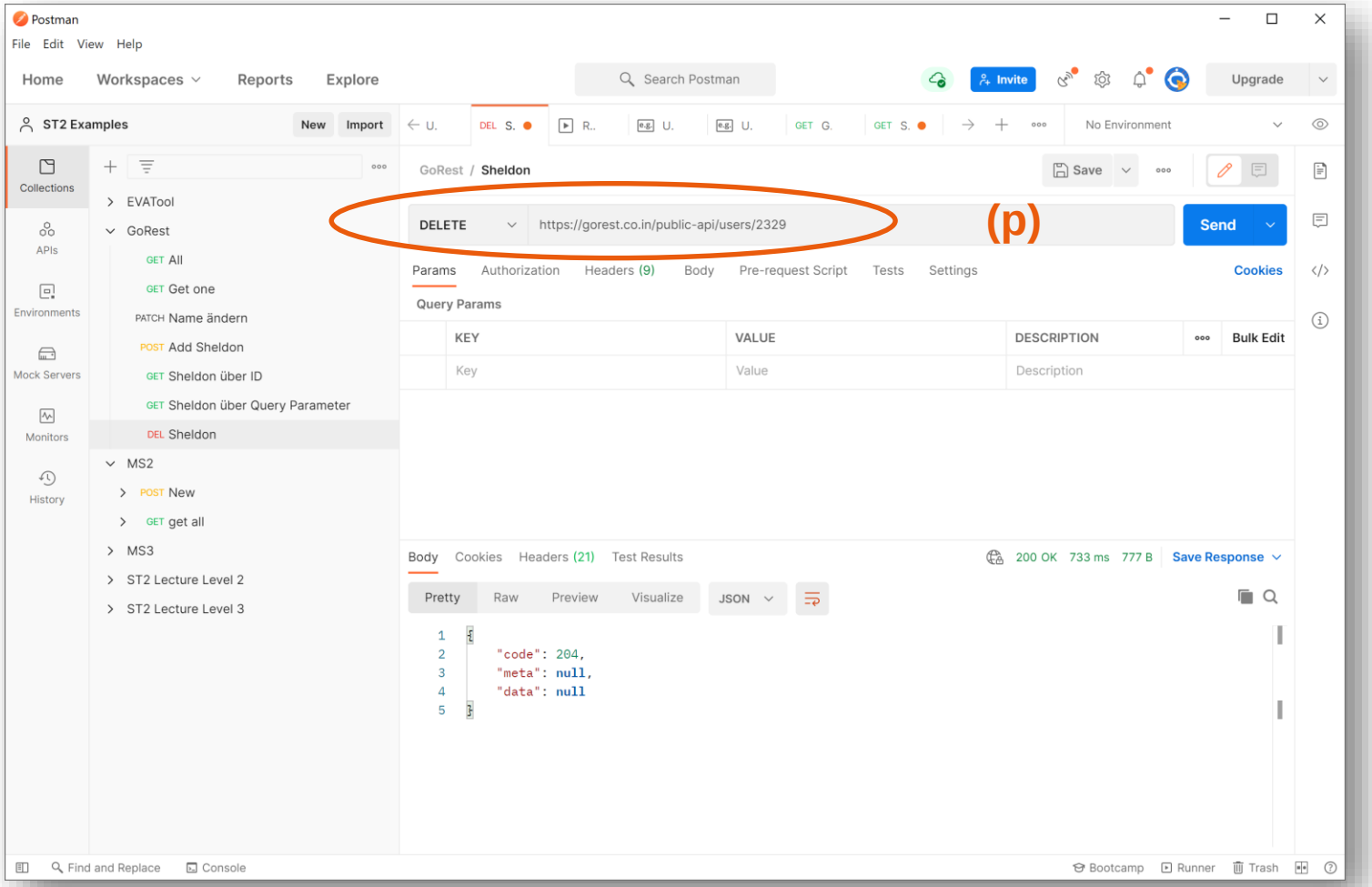
- Mit GET /users/2329 (der ID von Sheldon Cooper) (l) kann man eine neue Resource wieder abfragen.
- Die Antwort kommt im Request Body zurück (m).

Ausprobieren von Postman mit `gorest.co.in` (7)



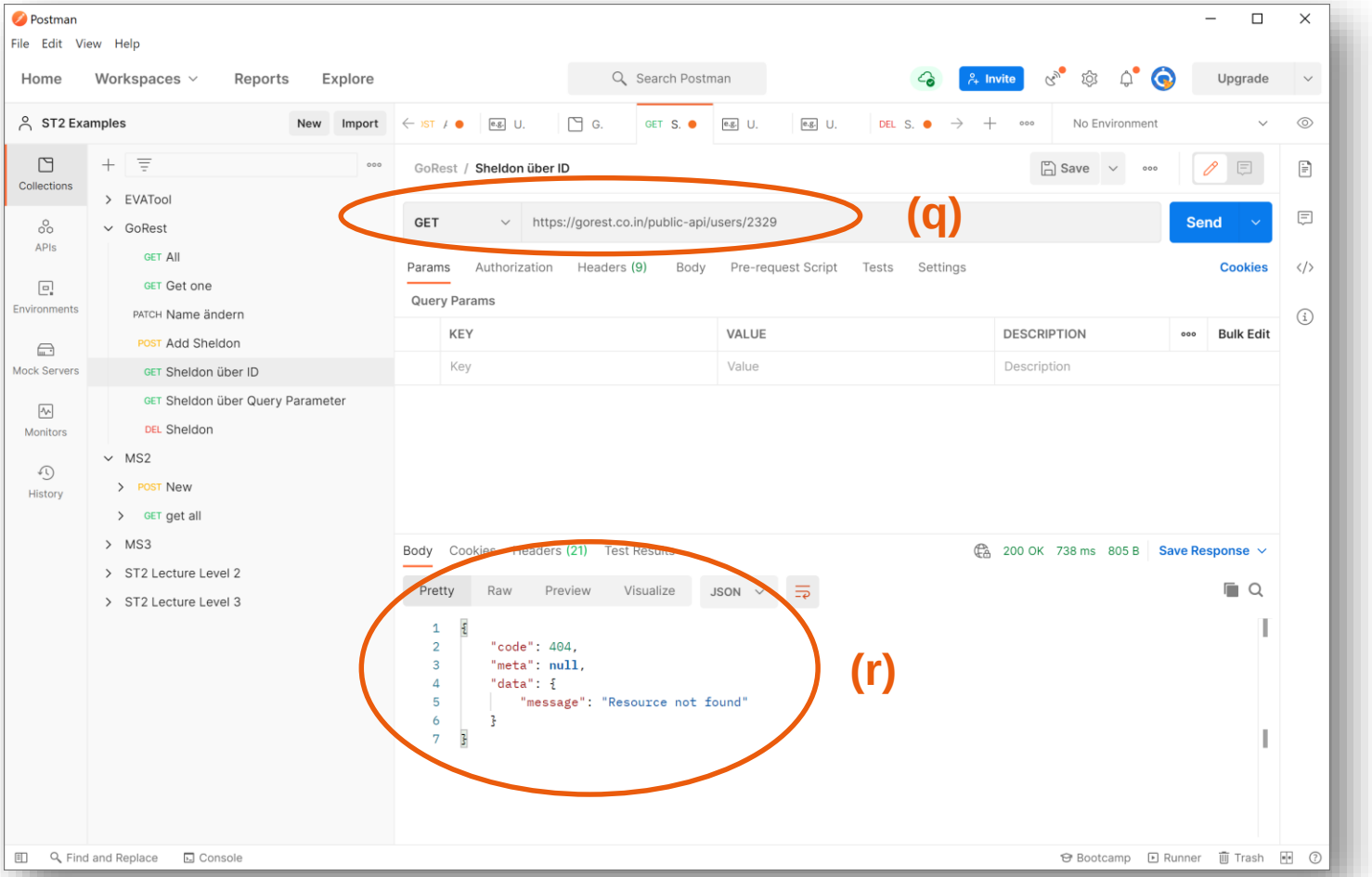
- Alternativ unterstützt dieses API auch die Abfrage von Query Parametern, hier “name” (n).
- Offensichtlich gibt es in der Datenbasis nur einen Sheldon Cooper (o).

Ausprobieren von Postman mit `gorest.co.in` (8)



- Mit DELETE kann man die Resource wieder löschen (p).

Ausprobieren von Postman mit gorest.co.in (9)



- Versucht man nun nochmal ein GET one mit der ID von Sheldon Cooper … (q).
- … kommt der Return Code 404 – “Resource not found” (r).

https://www.youtube.com/watch?v=gYlj8-0O-Y4

Prinzipien von REST

- REST = Representational State Transfer
- Architekturstil mit nachfolgenden Prinzipien (*)

(1) Kommunikation ist **zustandslos** (stateless)
(2) Ressourcen mit **eindeutiger Identifikation** (URI)
(3) Verwendung von **Standard-Methoden**
(4) Kommunikation unterstützt **Caching** (cacheable)
(5) Unterstützt verschiedene **Repräsentationen**
(6) Kommunikation über **Applikationszustand via Links** (*Hypermedia As The Engine Of Application State*, HATEOAS)

Quelle: zusammengefasst nach

- Fielding, Roy Thomas. “Architectural Styles and the Design of Network-Based Software Architectures.” University of California, Irvine, 2000, p. 77ff. http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf, abgerufen 20.6.2016
- Tilkov, Stefan, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*, p. 11ff. 3. Aufl. Heidelberg: dpunkt.verlag GmbH, 2015. (Im weiteren als [TilkovEtAl] referenziert)

Das REST-Maturity-Modell nach Richardson

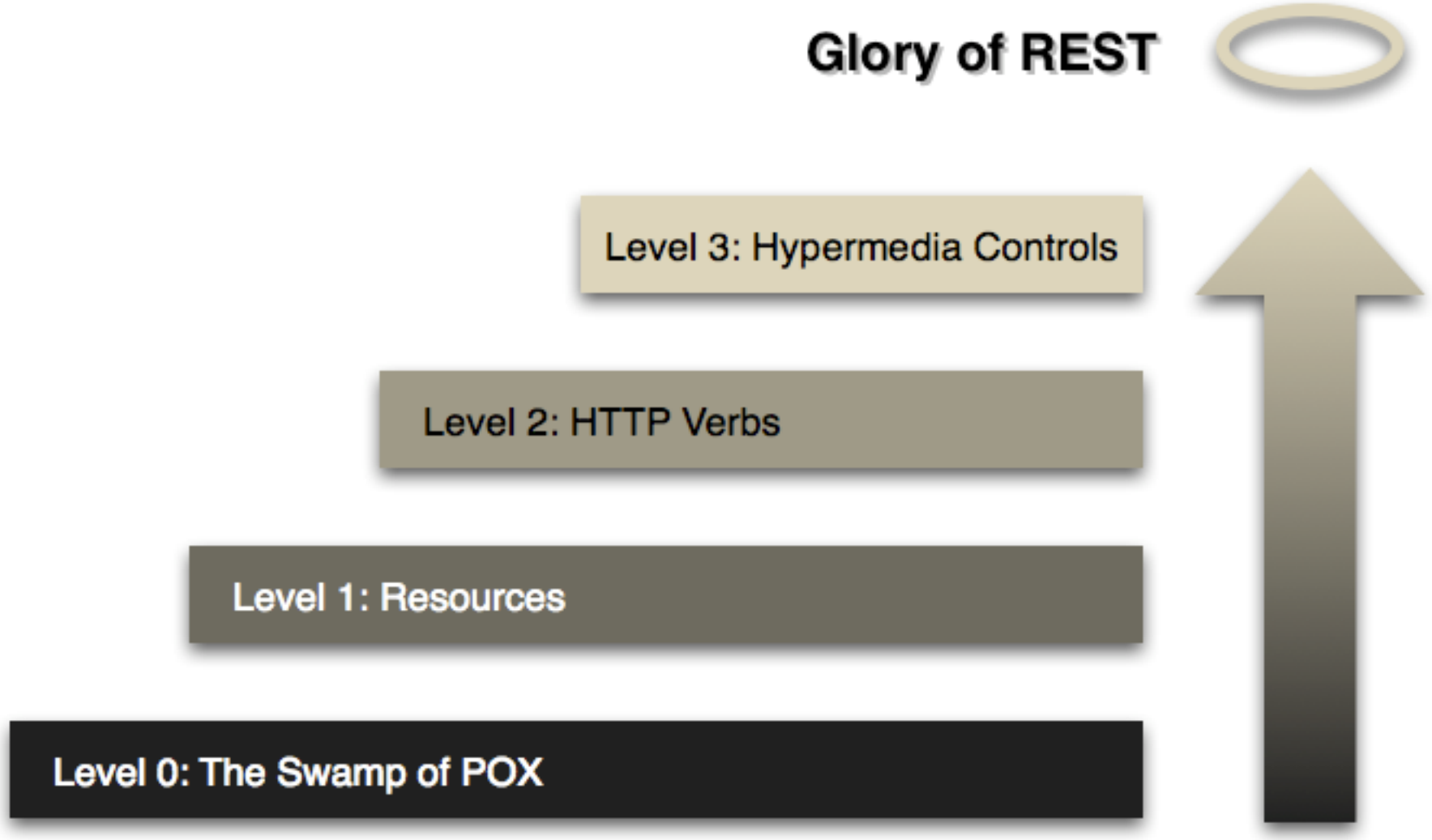


Bild: Martin Fowler,

Erfüllung der REST-Prinzipien in den Maturity Levels

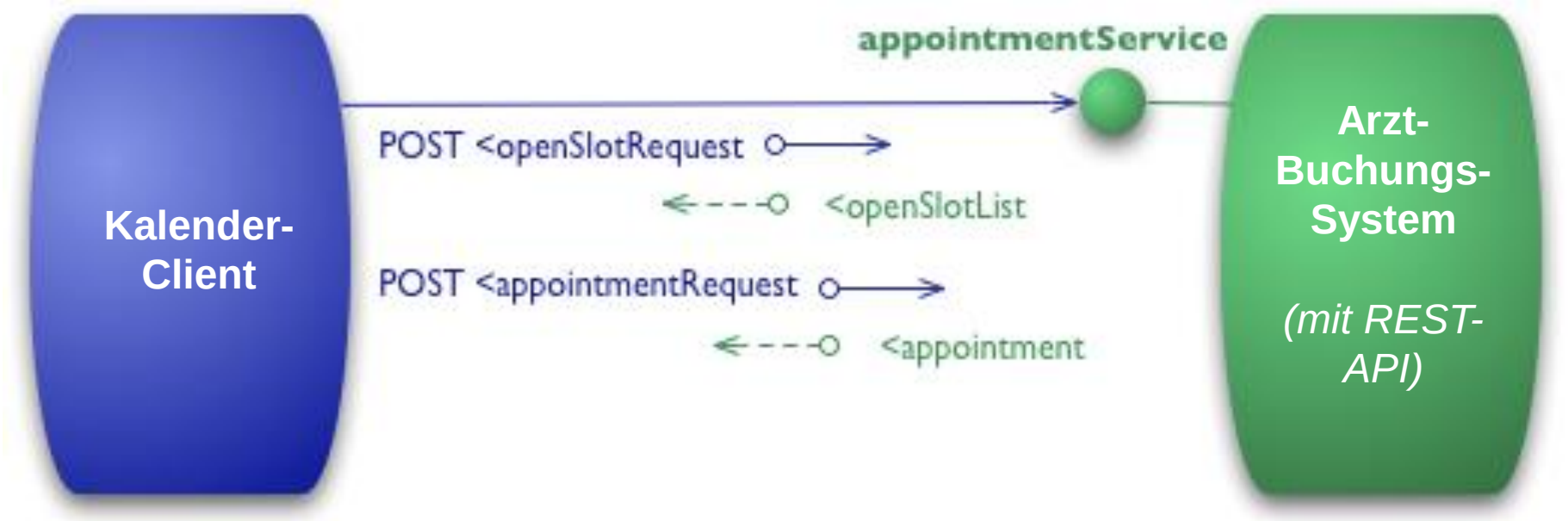
	Erfüllt in REST Maturity Level …?			
	0	1	2	3
(1) Kommunikation ist zustandslos (stateless)	ja	ja	ja	ja
(2) Ressourcen mit eindeutiger Identifikation (URI)	nein	ja	ja	ja
(3) Verwendung von Standard-Methoden	nein	nein	ja	ja
(4) Kommunikation unterstützt Caching (cacheable)	nein	nein	ja	ja
(5) Unterstützt verschiedene Repräsentationen	nein	nein	ja	ja
(6) Kommunikation über Applikationszustand via Links (HATEOAS)	nein	nein	nein	ja

Level 0: The Swamp of POX (Plain Old XML)

Bild: Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 0: The Swamp of POX (Plain Old XML) am Beispiel

- **Beispiel:** Arztbuchungssystem
- Unser REST-API soll es ermöglichen, aus einem beliebigen Kalenderclient heraus **Termine beim Arzt zu buchen**



Quelle Leicht gekürzt nach Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 0 – Beispiel konkret

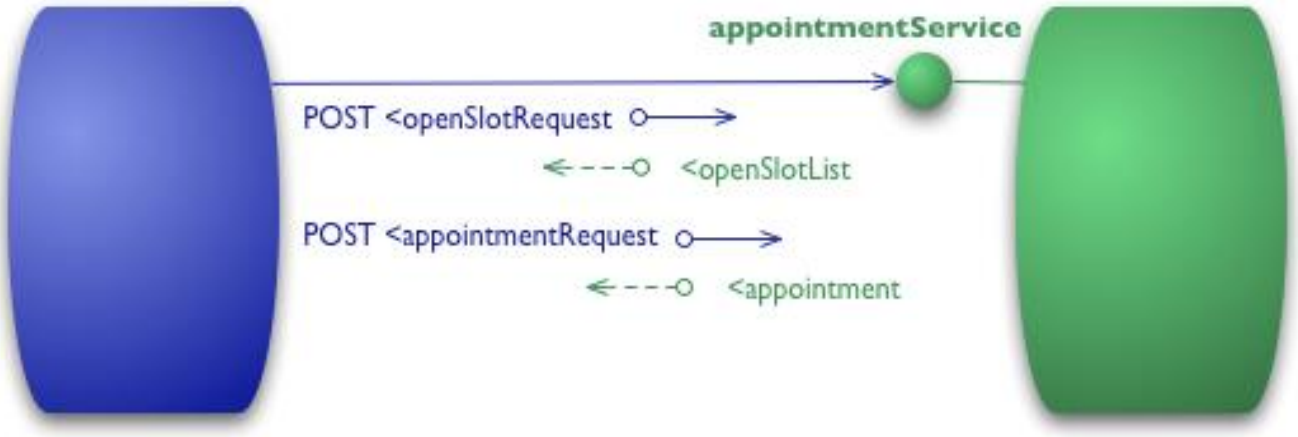
Anfrage nach Arzttermin

```
POST /appointmentService
HTTP/1.1

<openSlotRequest
        date="2010-01-04"
        doctor="mjones"/>
```

Antwort: Liste von Terminen

```
HTTP/1.1 200 OK

<openSlotList>
  <slot start = "1400„
        end = "1450"
        doctor_id = "mjones"/>
  <slot start = "1600"
        end = "1650"
        doctor_id = "mjones"/>
</openSlotList>
```

Bewertung von **Level 0: Swamp of POX (Plain Old XML)**



- Alle Interaktion hinter `/appointmentService` verborgen
- Details nur in gepostetem XML
- Schnittstellenaufbau wenig transparent

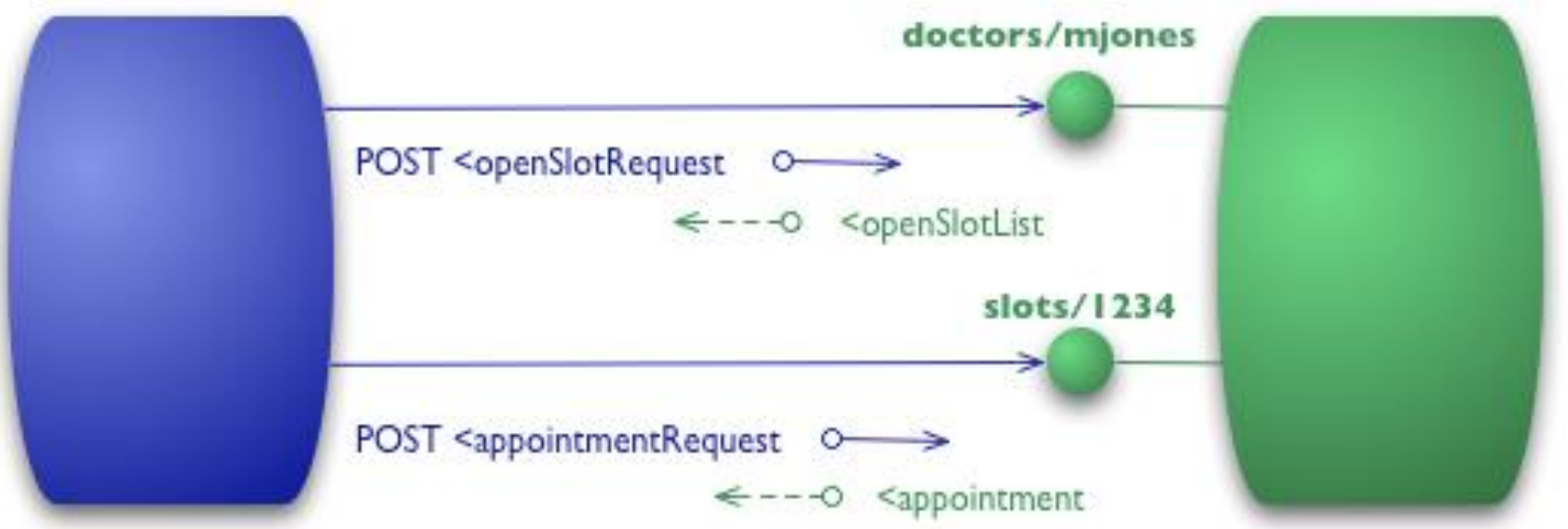
	Level 0	1	2	3
(1) Zustandslose Kommunikation	J			
(2) Ressourcen mit URI	N			
(3) Standard-Methoden	N			
(4) Cacheable	N			
(5) Verschiedene Repräsentationen	N			
(6) App.-zustand via Links	N			

Level 1: Resources

Bild: Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Quelle Leicht gekürzt nach Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 1: Resources



- Für offene Termine (open slots) und für Buchung (appointment request) jetzt jeweils eine dedizierte URI

Quelle Leicht gekürzt nach Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 1: Resources – Beispiel konkret

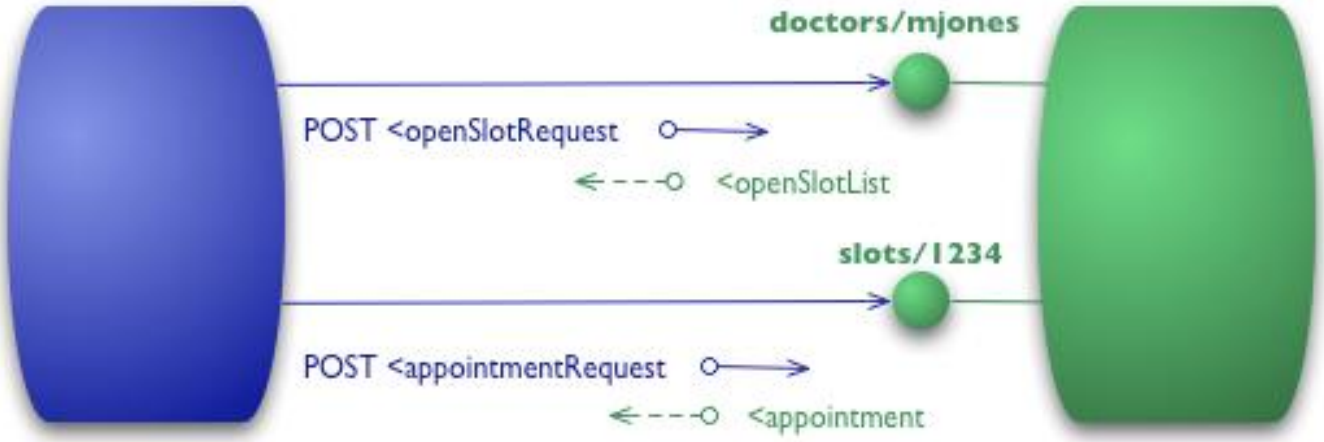
Anfrage freie Termine

```
POST /doctors/mjones HTTP/1.1
<openSlotRequest


HTTP/1.1 200 OK
<openSlotList>
  <slot id="1234"
        doctor="mjones"
        start="1400"
        end="1450"/>
  <slot id="5678" … />
</openSlotList>
```

Terminbuchung

```
POST /slots/1234 HTTP/1.1
<appointmentRequest
      patient="jsmith"/>

HTTP/1.1 200 OK
<appointment>
  <slot id="1234"
        doctor="mjones"
        start = "1400"
        end = "1450"/>
  <patient id="jsmith"/>
</appointment>
```

Bewertung von **Level 1: Resources**



- API ist durch URIs *etwas* transparenter
  - **ABER:** weiter nur POST genutzt
- => besser: HTTP-Verben zur Kennzeichnung des Request-Typs
  - GET, HEAD, PUT, POST, OPTIONS, DELETE

	Level 0	1	2	3
(1) Zustandslose Kommunikation	J	J		
(2) Ressourcen mit URI	N	J		
(3) Standard-Methoden	N	N		
(4) Cacheable	N	N		
(5) Verschiedene Repräsentationen	N	N		
(6) App.-zustand via Links	N	N		

Level 1: Resources

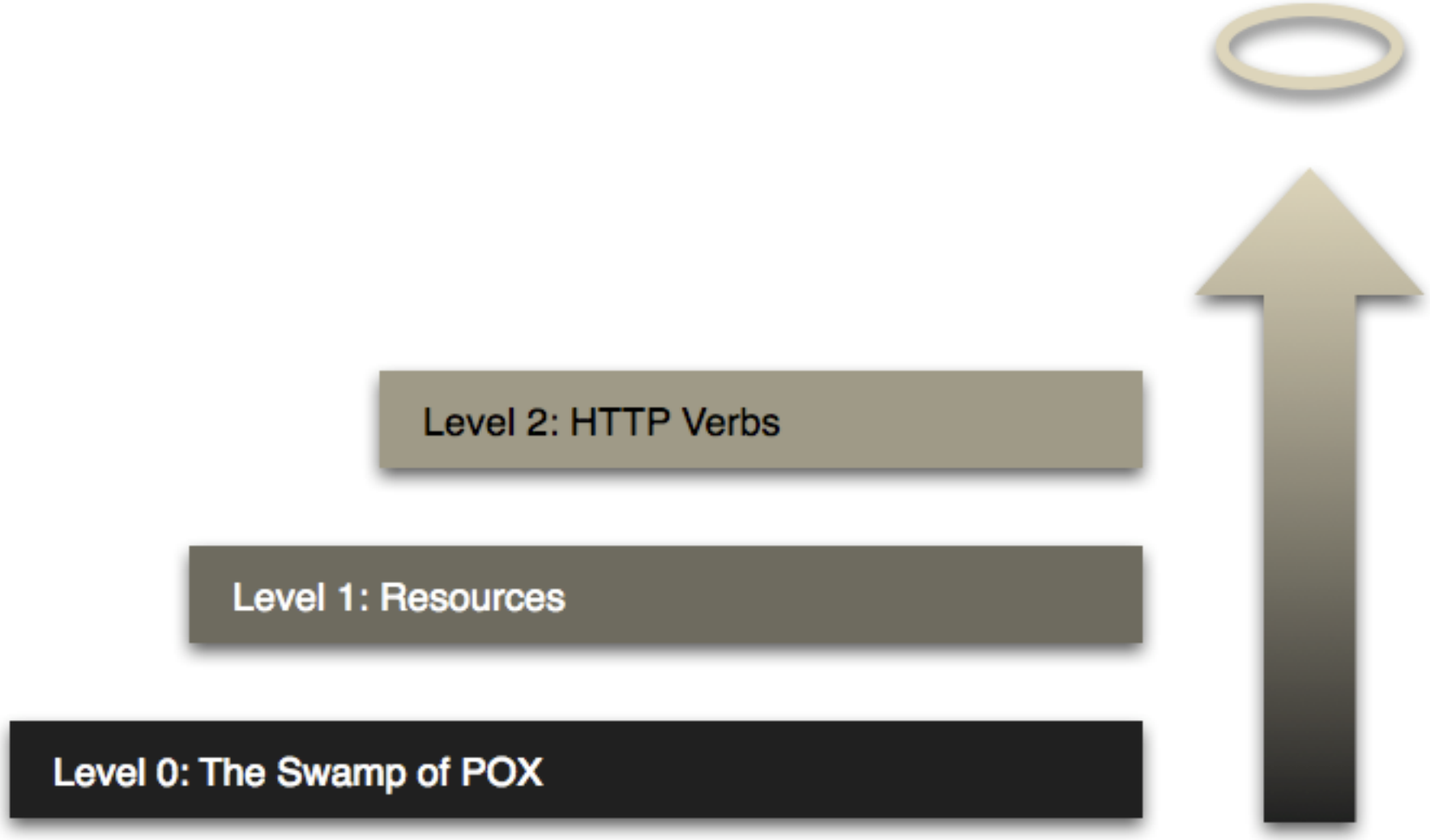


Bild: Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Mögliche Eigenschaften der HTTP Verben

- **sicher** =
  Ressourcen auf dem Server werden nicht verändert
- **idempotent** =
  mehrfacher Aufruf hat gleichen Effekt wie einmaliger Aufruf
- **identifizierbare Ressource** =
  Ressource ist aus der URI ersichtlich
- **Cache-fähig =**
  Ressource kann durch einen Cache schneller zur Verfügung gestellt werden
- **sichtbare Semantik** =
  Folge des Aufrufs aus dem Request ersichtlich

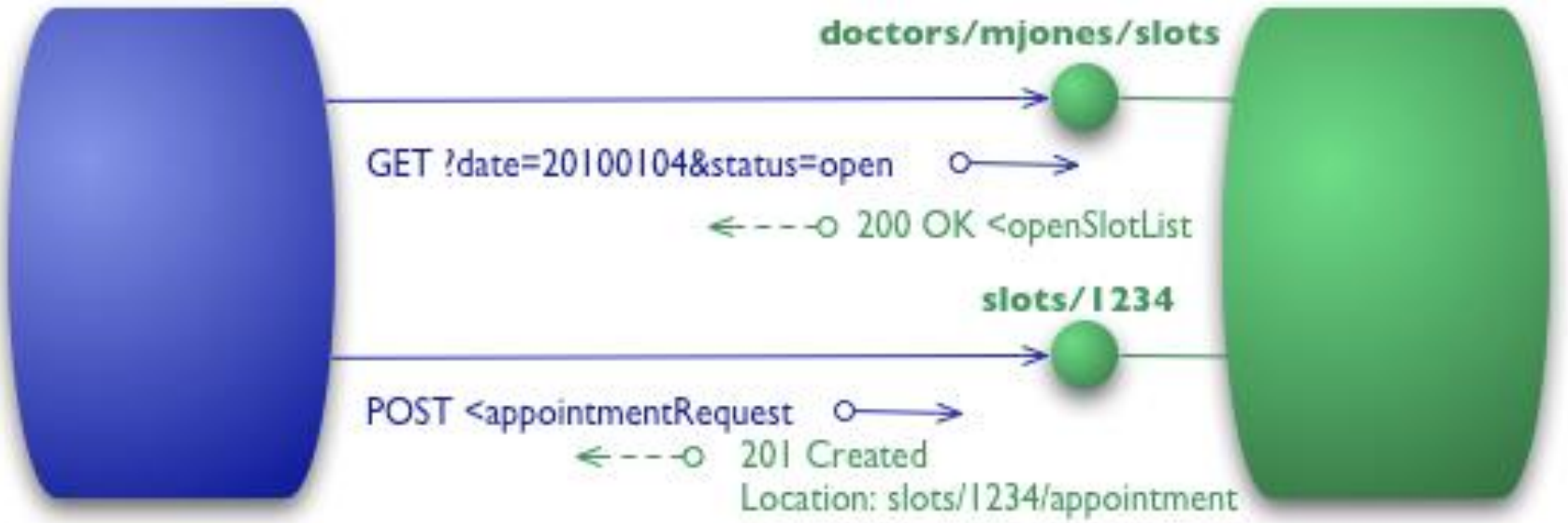
Quelle: [TilkovEtAl], p. 59

Bedeutung und Eigenschaften der HTTP Verben

Verb	Bedeutung	sicher	idem-potent	identifi-zierbare Ressource	Cache-fähig	sichtbare Semantik
GET	Anfordern einer Ressource durch Angabe einer URI	ja	ja	ja	ja	ja
HEAD	Wie **GET**, aber sendet nur den Header, nicht den Datenrumpf	ja	ja	ja	ja	ja
PUT	Hochladen einer Ressource unter einer angebenen URI. • Ressource schon da => Update • noch nicht da => neu erstellt	nein	ja	ja	nein	ja
PATCH	Partielles Ändern einer Ressource	nein	ja	ja	nein	ja
POST	Übermittlung von nicht näher spezifizierten Daten an den Server	nein	nein	nein	nein	nein
OPTIONS	Anfordern einer Liste der vom Server unterstützen Methoden und Merkmale	ja	ja	ja	nein	ja
DELETE	Löschen einer Ressource durch Angabe einer URI	nein	ja	ja	nein	ja

Quelle Leicht gekürzt nach Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 2: HTTP Verbs – am Beispiel



- Statt immer nur POST kennzeichnet jetzt das HTTP-Verb, um was für einen Request es sich handelt

Quelle Leicht gekürzt nach Martin Fowler, https://martinfowler.com/articles/richardsonMaturityModel.html

Level 2: HTTP Verbs – Beispiel konkret

Anfrage freie Termine

```
GET /doctors/mjones/slots?
   date=20100104&status=open
HTTP/1.1

HTTP/1.1 200 OK
<openSlotList>
  <slot id="1234"
        doctor="mjones"
        start="1400"
        end="1450"/>
  <slot id="5678" … />
</openSlotList>
```

Terminbuchung

```
POST /slots/1234 HTTP/1.1
<appointmentRequest
        patient="jsmith"/>

HTTP/1.1 201 OK
Location: slots/1234/appointment
<appointment>
  <slot id="1234"
        doctor="mjones"
        start = "1400"
        end = "1450"/>
  <patient id="jsmith"/>
</appointment>
```

Level 2: **Prinzip (3) Verwendung von Standard-Methoden**

Anfrage freie Termine

- HTTP-GET-Request an `/doctors/mjones/slots`
- Antwort nutzt HTTP Return Code
  - `HTTP/1.1 200 OK`
- Was, wenn es keinen Dr. Jones gibt?
  - `GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1`
  - `HTTP/1.1 404 Not found`

Terminbuchung

- erzeugt neue Ressource
  - `HTTP/1.1 201 OK`
    `Location: slots/1234/appointment`
- Erneutes Lesen mit GET, Änderungen mit PUT
- Was, wenn Zeitschlitz 1234 zwischenzeitlich vergeben wurde?
  - `HTTP/1.1 409 Conflict`

Häufiges REST-Antipattern: Verwechslung POST - PUT

POST

- die konkrete URI (insb. ID) ist **unbekannt**
  - URI wird erzeugt
  - Ressource wird angelegt
- Beispiel: Kunde neu anlegen
- *Request*
  - `POST /kunden?`
    `name="Müller, Max"`
- *Response*
  - `HTTP/1.1 201 OK`
  - `Location:/kunden/2615`

PUT

- die konkrete URI (insb. ID) ist **bekannt**
  - Ressource kann, muss aber nicht existierten
  - existiert: wird geändert
  - existiert nicht: wird angelegt
- Beispiel: Kundenname ändern
- *Request*
  - `PUT /kunden/2615?`
    `name="Müller-Vock, Max"`
- *Response*
  - `HTTP/1.1 200 OK`

Level 2: **Prinzip (4) Unterstützt Caching (cacheable)**

- Response auf auf einen GET-Request kann z.B. sein (Ausschnitt):

```
HTTP/1.1 200 OK
Expires: Mon, 29 May 2017 18:02:31 GMT
ETag: "HTTP/1.1
d979a0a78942b5bda05ace4214556a“
```

- Erneuter Aufruf des Clients schickt Etag mit
- Wenn sich die Resource nicht verändert hat, kann Server z.B. reagieren mit

```
HTTP/1.1 304 (Not Modified)
```

zum Nachlesen: https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html

Level 2: **Prinzip (5) Verschiedene Repräsentationen**

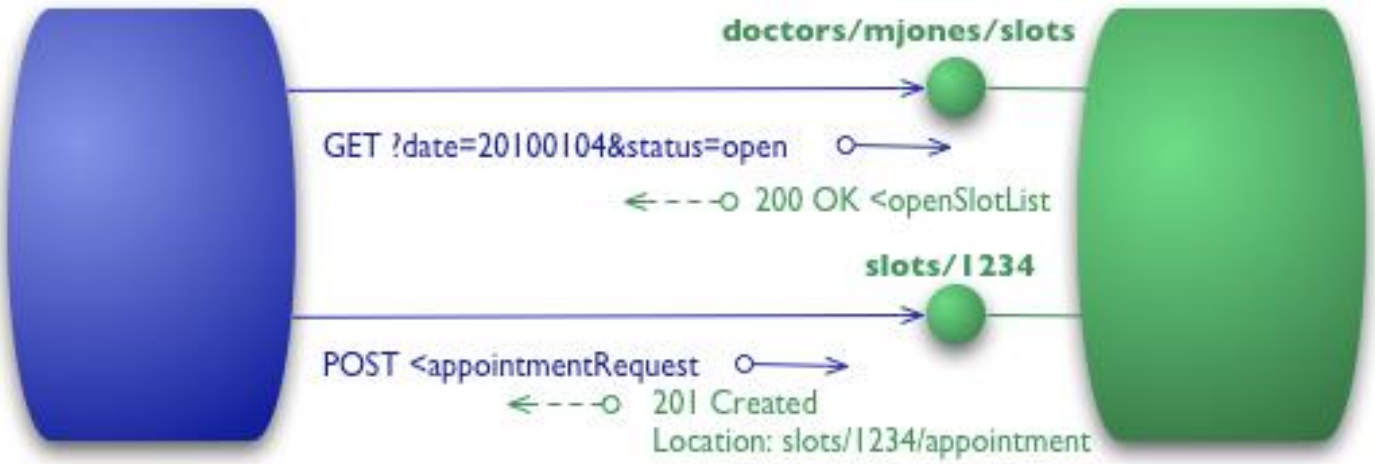
- Der Client kann die von ihm akzeptierten Formate angeben

```
GET /someURI HTTP/1.1
Accept: application/json, application/xml
```

- Der Server gibt das verwendete Format als MIME-Type an

```
HTTP/1.1 200 OK
Content-Type: application/json
```

Bewertung von **Level 2: HTTP-Verben**



- API ist jetzt deutlich …
  - transparenter
  - ausdruckstärker
- Fast alle REST-Prinzipien können damit umgesetzt werden
  - (siehe Beispiele nächste Seiten)

	Level 0	1	2	3
(1) Zustandslose Kommunikation	J	J	J	
(2) Ressourcen mit URI	N	J	J	
(3) Standard-Methoden	N	N	J	
(4) Cacheable	N	N	J	
(5) Verschiedene Repräsentationen	N	N	J	
(6) App.-zustand via Links	N	N	N	

https://www.youtube.com/watch?v=60pusIQ3nZ0

Stil-Regeln für URI-Design (1)

1. Forward slash separator (/) must be used to indicate a hierarchical relationship
2. A trailing forward slash (/) should not be included in URIs
   - **Falsch:** `http://canvas.org/shapes/`
   - **Richtig:** `http://canvas.org/shapes`
3. File extensions should not be included in URIs
   - **Falsch:** `http://college.org/students/3248234/address.json`
   - **Richtig:** `http://college.org/students/3248234/address`

Quelle: [Massé], p. 11ff. Beispiele leicht gekürzt.

Stil-Regeln für URI-Design (2)

4. Translation of entity names into URI

   Assume you have an entity **`MutualObligationAgreement`**

   a. ☺ Lower Camel Case: **`/mutualObligationAgreements/1234`**

   b. ☺ Kebab (Spinal) Case: **`/mutual-obligation-agreements/1234`**

- Was man **nicht** macht: Underscores!
  - ☹ `/mutual_obligation_agreement/…`

- Es gibt hier keine “harten” Regeln
  - In der Praxis gibt es beides
  - Im Praktikum verwenden wir 5.a. (Lower Camel Case)
  - Wie immer Sie es machen: Machen Sie es konsistent!

Aggregates => REST-APIs

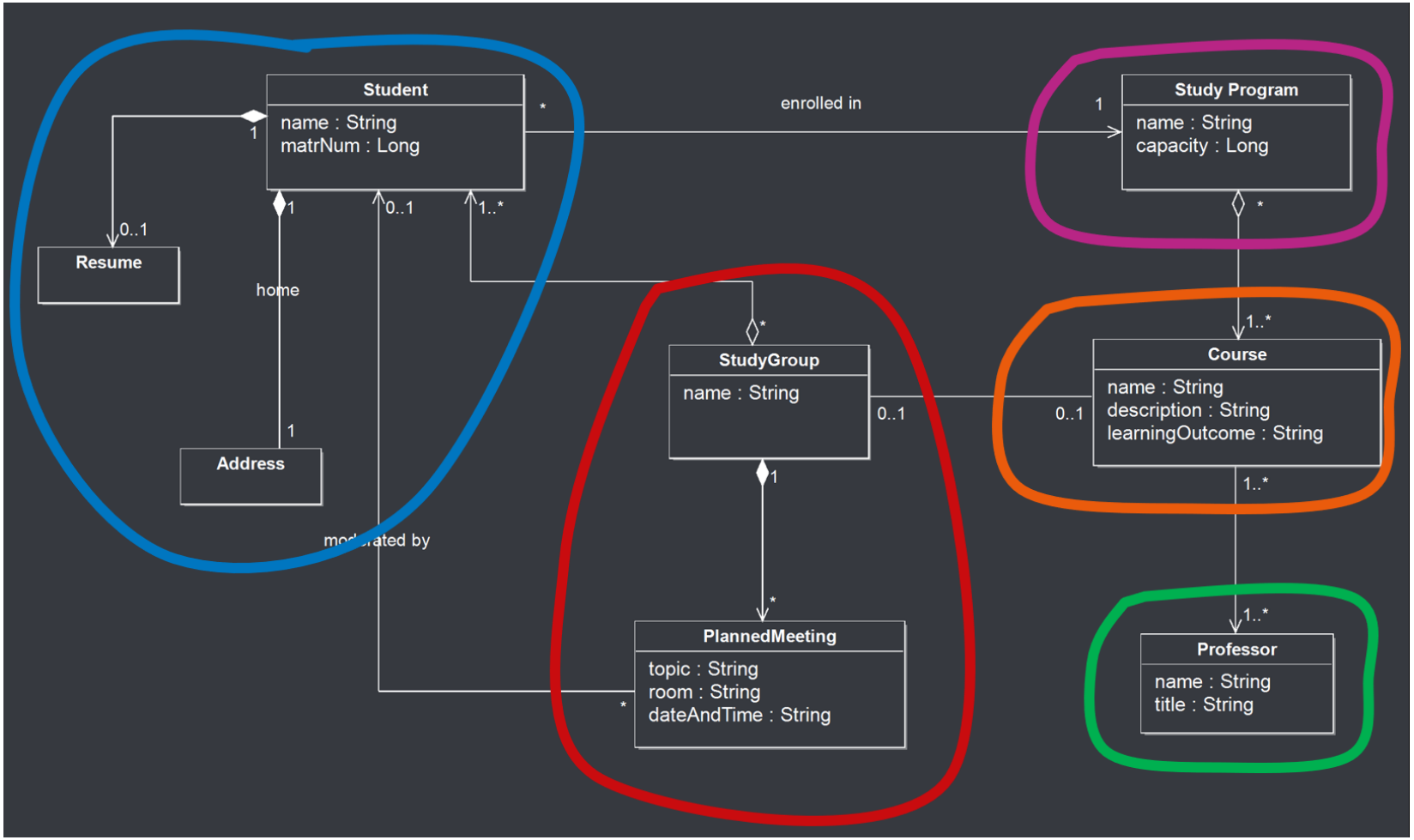
Endpoint

=

Aggregate

- … und es gelten alle Aggregate-Regeln
- z.B. keine Möglichkeit des direkten Zugriffs auf innere Entities
- siehe nächste Seiten

Ein kleines Beispiel-Datenmodell



- Hier unser Beispiel für die Spezifikation von REST-Endpoints – das Campus-Management-System.
- Farbig umkringelt: Die Aggregates.

Aufbau der URIs in REST-APIs

- URIs bilden Hierarchien in den Daten ab
  - Nicht alle Pfade durchs Datenmodell müssen im REST-API verfügbar sein!
- Drei Kernelemente in der URI:

	URI-Spezifikation Konkrete Beispiel-URL
1. Collections ○ Sammlungen von Ressourcen ○ im Plural benannt	**/students** https://hs-mgmt.de/students
2. IDs ○ Selektieren Entity aus Collection ○ Kann Zahl oder Text sein	**/students/{s-id}** https://hs-mgmt.de/students/123456
3. Entities ○ Steht für **einzelne** Ressource – Annahme: Student => Address 1:1 ○ im Singular benannt	**/students/{s-id}/address** https://hs-mgmt.de/students/123456/address

Anwendung REST-Verben (1)

- HTTP-Verben werden auf das **letzte (rechteste) Element** des URI-Pfads angewendet
- Sie wirken **bezüglich des jeweiligen Parent**

- `GET /studyPrograms`
  - gebe alle Studiengänge zurück *(zu entscheiden: will man das im API?)*
- `GET /studyPrograms/{s-id}`
  - gebe einen bestimmtes SG zurück
- `GET /studyPrograms/{s-id}/courses`
  - gebe alle Fächer zu einem Studiengang zurück

- `DELETE /studyPrograms`
  - lösche alle Studiengänge *(will man das im API?)*
- `DELETE /studyPrograms/{s-id}/courses`
  - lösche alle Zuordnungen von Fächern zu einem Studiengang *(ist das gewollt?)*
    - Wirkt auf Parent => nur Beziehung wird gelöscht, die Fächer selbst bleiben bestehen

Anwendung REST-Verben (2)

- **`POST /studyPrograms`**
  - Lege neuen Studiengang an
    - System legt neuen Studiengang an
    - Details des Studiengangs müssen im Request-Body mit übertragen werden
    - z.B. als JSON oder XML

- **`PUT /studyPrograms/{s-id}/courses/{c-id}`**
  - Ordne ein Fach einem Studiengang zu
    - Fach muss schon existieren

- **`PATCH /studyPrograms/{s-id}`**
  - Mache einen Update des Studiengangs
    - Details (z.B. neuer Name) muss im Request-Body übertragen werden

Query-Design

Query String

```
GET /mypath?att1=value1&att2=value2
```

- Folgende Operationen werden im Query-String ausgedrückt:
  - Searching
  - Paging
  - Filtering
  - Sorting

- Beispiel: Filtern
- *alle Studierenden mit Matrikel-Nummern, die mit 1,2 oder 3 beginnen*

```
GET /students?
        matrNumFrom=10000000&matrNumUntil=39999999
```

Aggregate

- ~~`PUT /plannedMeetings/{p-id}`~~
  - Kein direkter Zugriff auf inneres Entity mit globaler ID

- ~~`PUT /studyGroups/{s-id}/ plannedMeetings/{p-id}`~~
  - Kein Zugriff auf inneres Entity mit globaler ID
    - auch wenn es über das Aggregate Root geht

- `PUT /studyGroups/{s-id}/ plannedMeetings/{local-id}`
  - z.B.
    `PUT /studyGroups/123456/ plannedMeetings/ST2`
  - Zugriff auf inneres Entity mit **lokaler** ID ist ok
    - Annahme: Topic ist die lokale ID – es gibt nur ein Meeting pro Topic

HTTP Return Codes

in REST APIs

Kein Dr. Jones?

404

NOT FOUND

Technology
Arts Sciences
TH Köln

https://www.youtube.com/watch?v=OvJlhQw14Vg

Kategorien von HTTP Response Status Codes

- **1xx**: Informational
  - Communicates transfer protocol-level information.
- **2xx**: Success
  - Indicates that the client's request was accepted successfully.
- **3xx**: Redirection
  - Indicates that the client must take some additional action in order to complete their request.
- **4xx**: Client Error
  - This category of error status codes points the finger at clients.
- **5xx**: Server Error
  - The server takes responsibility for these error status codes.

Quelle: [Massé], p. 28

„Best of“ - Häufig genutze HTTP Response Status Codes

- **2xx**: Success
  - **200: OK** - Allgemeiner Erfolg
  - **201: Created** – Ressource oder Beziehung angelegt, z.B. nach POST
  - **204: No Content** – alles gut gegangen, aber es wird kein Inhalt zurückgeliefert
- **3xx**: Redirection
  - **301: Moved Permanently** – API redesigned, neue URL im Response Location Header
  - **304: Not Modified** – ETag im Request => kein neues Senden nötig
- **4xx**: Client Error
  - **400: Bad Request** – i.d.R. Syntax Error, z.B. in JSON/XML Payload
  - **401: Unauthorized** – Zugriff wäre möglich, Client ist aber **nicht authentifiziert**
  - **403: Forbidden** – Client **hat Credentials**, aber die falschen => neuer Versuch sinnlos
  - **404: Not Found** – Ressource ist gar nicht da
    - (oder man will dem Client nicht verraten, dass er sie nicht sehen darf!)
  - **405: Method Not Allowed** – gewünschtes **HTTP-Verb** hier nicht unterstützt
  - **406: Not Acceptable** – gewünschtes **Ergebnisformat** des Clients hier nicht unterstützt
  - **409: Conflict** – z.B. Arztsystem: Client versucht, einen besetzten Termin zu buchen
  - **422: Unprocessable Entity** – Semantischer Fehler bei Verarbeitung des Request Bodies

Quelle (und zum selbst weiter recherchieren): überall im Netz, z.B. https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications

Klärung ob Return Code passt – Nachlesen in der RFC

Ist 409 wirklich der richtige Return Code, wenn man einen Termin buchen will, der zwischenzeitlich vergeben wurde?

https://datatracker.ietf.org/doc/html/rfc7231#page-60

6.5.8. 409 Conflict

> The 409 (Conflict) status code indicates that the request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request. The server SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

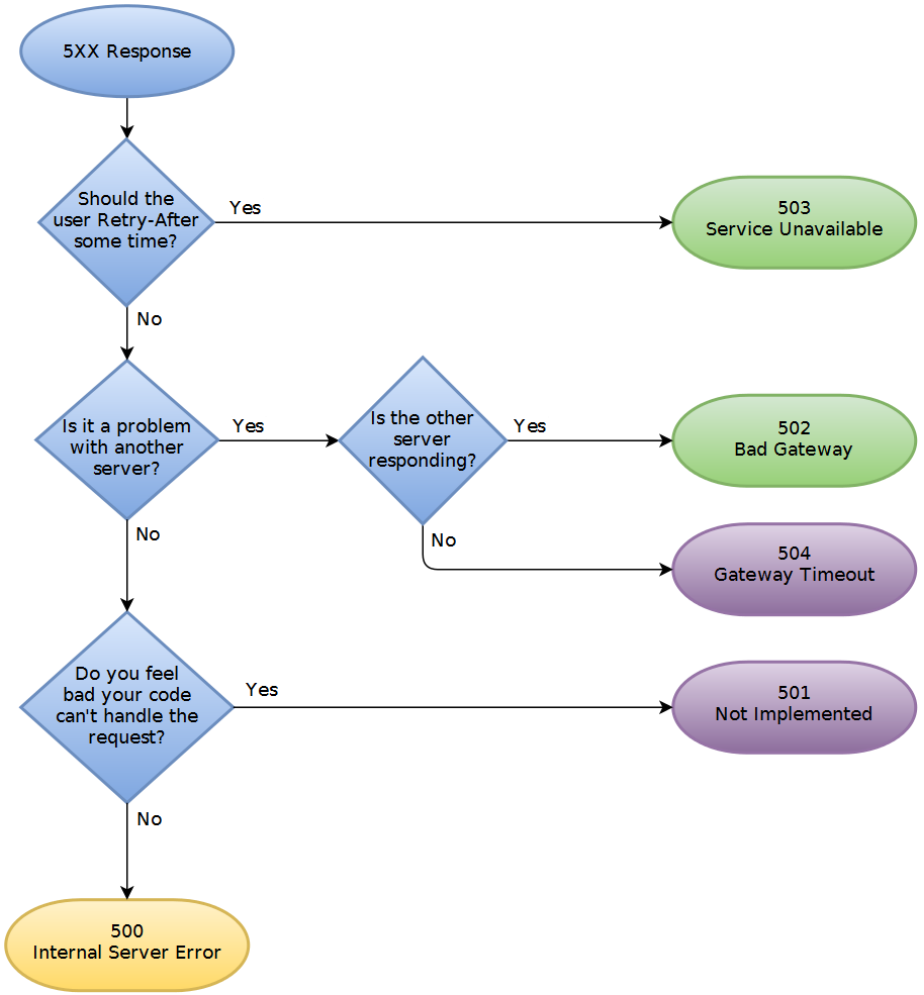
=> ja, 409 ist für den Fall richtig!

„Best of“ - Häufig genutze HTTP Response Status Codes

- **5xx**: Server Error
  - **500: Internal Server Error** – irgendwas auf Server-Seite geht schief

Sehr schönes
Entscheidungsdiagramm
zum selbst weiter recherchieren
(Ausschnitt rechts)

https://www.codetinkerer.com/2015/12/04/choosing-an-http-status-code.html



Drei Fehlerbeispiele

- Fall 1: Die angefragte Ressource existiert nicht.
  - `GET /studyGroups/0`
    `// die ID gibt es nicht`

404

- Fall 2: Verbotene Verben
  - `DELETE /studyPrograms/12345`
    `// Studiengänge darf man nicht löschen`

405

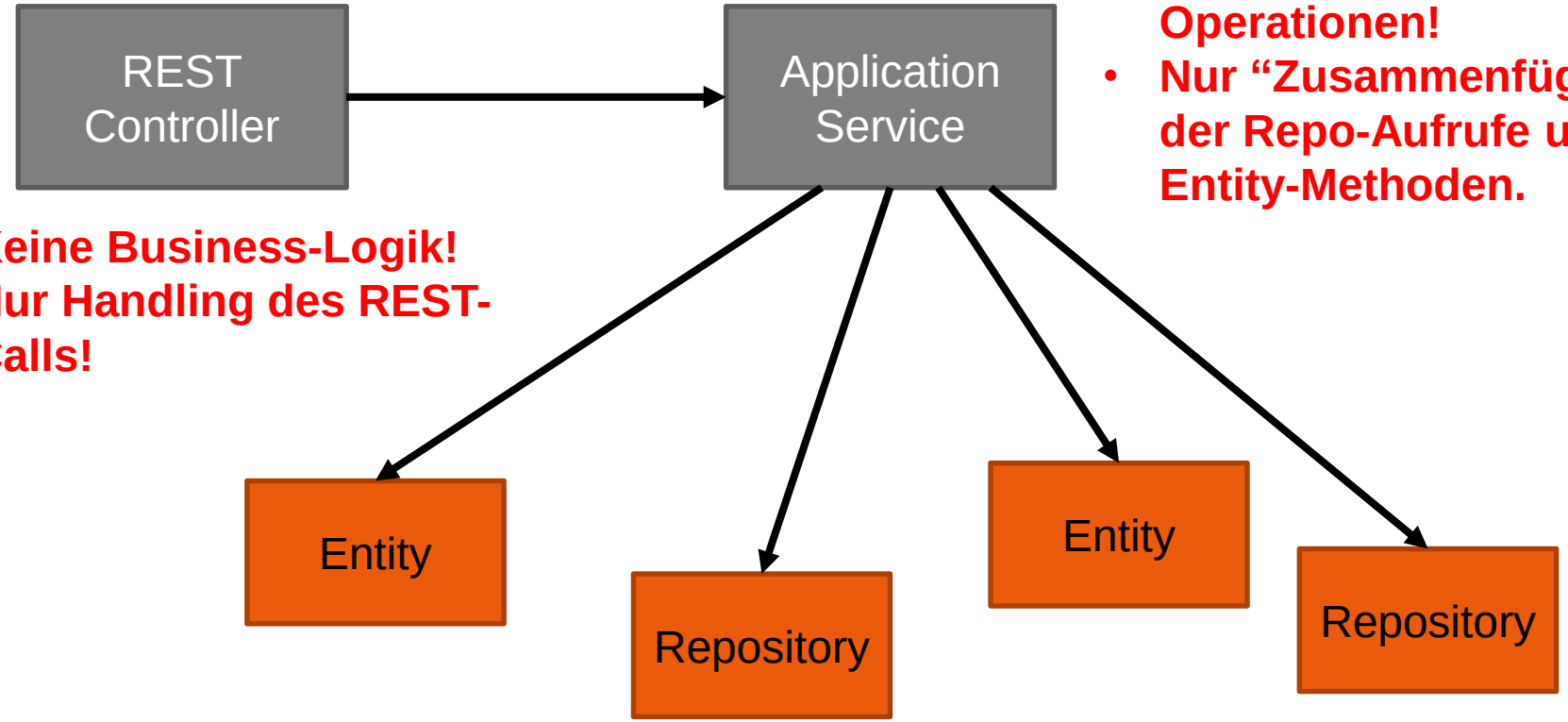
- Fall 3: Verletzung von Invarianten
  - `POST /studyGroups/54321/plannedMeetings`
    `// die hinzugefügten Meetings verletzen die Invariante:`
    `// es gibt jetzt mehrere Meetings zum gleichen Thema`

409

Coding REST APIs mit Spring Web MVC

(vorläufiger Scriptteil, Video dazu folgt)

Software Pattern für das Coding von REST APIs



- **Keine Business-Logik!**
- **Nur Handling des REST-Calls!**

- **Keine Domänen-Operationen!**
- **Nur “Zusammenfügen” der Repo-Aufrufe und Entity-Methoden.**

Aufbau eines Spring REST Controllers

```
@RestController
public class StudyProgramController {
   // autowire application services here

   @GetMapping("/studyPrograms")
   public Iterable<StudyProgramDto> getAllStudyPrograms() {
      return studyProgramApplicationService.findAll( minCapacity, maxCapacity );
   }

   @GetMapping("/studyPrograms/{id}")
   public ResponseEntity<StudyProgramDto> getOneStudyProgram( @PathVariable UUID id ) {
     return new ResponseEntity(studyProgramApplicationService.findById( id ), OK );
   }

   @DeleteMapping("/studyPrograms/{id}")
   public ResponseEntity<StudyProgramDto> deleteOneStudyProgram( @PathVariable UUID id ) {
     studyProgramApplicationService.delete( id );
     return new ResponseEntity( HttpStatus.NO_CONTENT );
   }
   ...
}
```

GET all

```java
/**
 * This API doesn't really make sense - it is just a "Hello World"
 */
@GetMapping("/studyGroups/helloWorld/{id}")
public ResponseEntity<String> getStudyGroupHelloWorld(@PathVariable String id) {
  return new ResponseEntity<String>( "Hello World " + id, HttpStatus.I_AM_A_TEAPOT );
}
```

1

StudyGroupController

```java
//@GetMapping("/studyGroups") - the VERY simple version
public Iterable<StudyGroup> getAllStudyGroupsVerySimple() {
  return studyGroupRepository.findAll();
}
```

2

```java
@GetMapping("/studyGroups")
public Iterable<StudyGroupDto> getAllStudyGroups() {
  Iterable<StudyGroup> found = studyGroupRepository.findAll();
  List foundDtos = new ArrayList<StudyGroupDto>();
  ModelMapper modelMapper = new ModelMapper();
  for ( StudyGroup sg : found ) {
    foundDtos.add( modelMapper.map( sg, StudyGroupDto.class ) );
  }
  return foundDtos;
}
```

3

- Hier sieht man, wie REST-Controller geschrieben werden.

1. Das erste Beispiel ist zu nichts nutze, nur ein „Hello World“. Es gibt kein valides JSON zurück. Daher auch der Witz-Return-Code 418.
2. Das zweite Beispiel zeigt ein „Get All“ in der aller-simpelsten Form. Ein einfacher Repo-Aufruf genügt, der Returncode wird automatisch von Spring gesetzt.
3. Das dritte Beispiel ist so, wie man es machen sollte: Immer ein DTO verwenden, damit man den Inhalt seines Response im eigenen API steuern kann.

GET One

Unit Test

```
/**
 * GET /studyPrograms/{s-id}
 */
@Test
public void testGetOneStudyProgram() throws Exception {
  mockMvc .perform(get("/studyPrograms/" + INF_MASTER_ID ))
      .andDo(print())
      .andExpect(status().isOk())
      .andExpect(jsonPath("$.name", is(INF_MASTER_NAME)));
}
```

StudyProgramController

```
@GetMapping("/studyPrograms/{id}")
public ResponseEntity<StudyProgramDto> getOneStudyProgram( @PathVariable UUID id ) {
  Optional<StudyProgram> found = studyProgramRepository.findById( id );
  if( found.isEmpty() )
    return new ResponseEntity<>( HttpStatus.NOT_FOUND );
  else {
    return new ResponseEntity<StudyProgramDto>(
        modelMapper.map( found.get(), StudyProgramDto.class ), HttpStatus.OK );
  }
}
```

- Das Get One folgt dem oben dargestellen Muster.
- Zunächst immer eine Prüfung, ob es die ID gibt (findById Methode im Repo).
  - Das Optional ist leer, wenn es die ID nicht gibt. In dem Fall wird 404 zurückgegeben, sonst das gefundene Entity.
- Vor der Rückgabe wird das Entity noch auf das DTO gemappt, um den gewünschten API-View zu erzielen.

DELETE One

```
/**
 * DELETE /studyPrograms/{s-id}
 */
@Test
public void testDeleteOneStudyProgramIsSuccessful() throws Exception {
  mockMvc .perform(delete("/studyPrograms/" + INF_MASTER_ID ))
      .andDo(print())
      .andExpect(status().isNoContent());
  mockMvc .perform(get("/studyPrograms/" + INF_MASTER_ID ))
      .andDo(print())
      .andExpect(status().isNotFound());
}
```

Unit Test

StudyProgramController

```
@DeleteMapping("/studyPrograms/{id}")
public ResponseEntity<StudyProgramDto> deleteOneStudyProgram( @PathVariable UUID id ) {
  Optional<StudyProgram> found = studyProgramRepository.findById( id );
  if( found.isEmpty() )
    return new ResponseEntity<>( HttpStatus.NOT_FOUND );
  else {
    studyProgramRepository.delete( found.get() );
    return new ResponseEntity( HttpStatus.NO_CONTENT );
  }
}
```

- Delete One wird ganz ähnlich zu Get One implementiert.
- Beim Test nutze ich einen anschließenden GET One Call, um zu prüfen, ob es auch wirklich gelöscht ist.

DELETE All: Verhindern eines Verbs

```
/**
 * Illegal DELETE all
 * DELETE /studyPrograms
 */
@Test
public void testItIsIllegalToDeleteAllStudyPrograms() throws Exception {
  mockMvc .perform( delete("/studyPrograms" ) )
       .andDo(print())
       .andExpect(status().isMethodNotAllowed());
}
```

Unit Test

- Ein Delete All wird häufig in APIs verboten sein.
- Hier reicht es, einfach kein Mapping anzubieten – Spring gibt automatisch dann 405 (not allowed) zurück. Man braucht in dem Fall also einfach nichts zu implementieren.

POST (1)

StudyProgramController

```
//@PostMapping("/studyPrograms")
public StudyProgram createNewStudyProgramSimple( @RequestBody StudyProgram studyProgram ) {
  return studyProgramRepository.save(studyProgram);
}
```

```
/**
 * POST /studyPrograms
 */
@Test
public void testCreateNewStudyProgram() throws Exception {
  StudyProgramDto studyProgramDto = new StudyProgramDto();
  studyProgramDto.setName( "Medieninformatik" );
  ObjectMapper objectMapper = new ObjectMapper();
  String json = objectMapper.writeValueAsString(studyProgramDto);

  MvcResult result = mockMvc.perform( post("/studyPrograms")
                    .contentType(APPLICATION_JSON).content(json))
      //.andExpect(status().is2xxSuccessful())
      .andExpect(status().isCreated())
      .andExpect(jsonPath("$.name", is("Medieninformatik")))
      .andReturn();

  // check if the object is indeed there, by sending a GET call
  String location = result.getResponse().getHeader("Location");
  mockMvc .perform(get( location ))
      .andDo(print())
      .andExpect(status().isOk());
}
```

Unit Test

- Post könnte man relativ leicht implementieren (s.o.)
- Aber damit verliert man drei Dinge:
  1. Die ID des erzeugten Objects muss eigentlich in den Header des Response – das passiert hier nicht.
  2. Besser man akzeptiert nur DTOs, dann können einem keine Daten geändert werden, die man nicht ändern will. Das geht bei dieser simplen Variante nicht.
  3. Der Return-Code ist nicht differenziert (besser 201, created)

POST (2)

StudyProgramController

```
@PostMapping("/studyPrograms")
public ResponseEntity<?> createNewStudyProgram( @RequestBody StudyProgramDto studyProgramDto) {
  try {
    // create new so that the ID has been set, then transfer the received DTO
    StudyProgram newStudyProgram = new StudyProgram();
    modelMapper.map( studyProgramDto, newStudyProgram );
    studyProgramRepository.save( newStudyProgram );
    URI returnURI = ServletUriComponentsBuilder
        .fromCurrentRequest()
        .path("/{id}")
        .buildAndExpand(newStudyProgram.getId())
        .toUri();
    return ResponseEntity
        .created(returnURI)
        .body( studyProgramDto );
  }
  catch( Exception e ) {
    // something bad happened -
    return new ResponseEntity<>( HttpStatus.UNPROCESSABLE_ENTITY );
  }
}
```

- Diese Variante behebt die genannten Probleme.
- Besonders das Übertragen der ID in den Header ist wichtig, sonst weiß ein Client gar nicht, **welches** Objekt jetzt erzeugt wurde.

PUT (1)

```
/**
 * PUT /studyPrograms/{s-id}
 * Case 1: Entity is new
 */
@Test
public void testPutToCreateNewEntityStudyProgram() throws Exception {
  StudyProgramDto justSomethingDto = new StudyProgramDto();
  String newName = "Just Something, Bachelor";
  justSomethingDto.setName( newName );
  UUID id = UUID.randomUUID();
  ObjectMapper objectMapper = new ObjectMapper();
  String json = objectMapper.writeValueAsString(justSomethingDto);

  MvcResult result = mockMvc.perform( put("/studyPrograms/" + id )
      .contentType(APPLICATION_JSON).content(json))
      .andExpect(status().isCreated() )
      .andReturn();

  Optional<StudyProgram> found = studyProgramRepository.findById( id );
  assertTrue( "Expected entity with ID " + id + " to be there", !found.isEmpty() );
  assertTrue( "Expected name " + newName, found.get().getName().equals( newName ) );
  studyProgramRepository.deleteById( id );
}
```

Unit Test

- Test von Put ist etwas kompliziert, weil zwei Varianten getestet werden müssen: Entity ist neu (wie hier) …

PUT (2)

```
/**
 * PUT /studyPrograms/{s-id}
 * Case 2: Entity already exists
 */
@Test
public void testPutToUpdateExistingEntity() throws Exception {
  StudyProgram inf = studyProgramRepository.findById( INF_MASTER_ID ).get();
  ModelMapper modelMapper = new ModelMapper();
  StudyProgramDto infDto = modelMapper.map( inf, StudyProgramDto.class );
  String newName = "Power Gaming Master";
  infDto.setName( newName );
  ObjectMapper objectMapper = new ObjectMapper();
  String json = objectMapper.writeValueAsString(infDto);

  MvcResult result = mockMvc.perform( put("/studyPrograms/" + inf.getId() )
      .contentType(APPLICATION_JSON).content(json))
      .andExpect(status().isNoContent() )
      .andReturn();

  Optional<StudyProgram> found = studyProgramRepository.findById( inf.getId() );
  assertTrue( "Expected entity with ID " + inf.getId() + " to be there", !found.isEmpty() );
  assertTrue( "Expected new name " + newName, found.get().getName().equals( newName ) );
}
```

Unit Test

- … oder Entity existiert schon.

PUT (3)

```
@PutMapping("/studyPrograms/{id}")
public ResponseEntity changeOneStudyProgram( @PathVariable UUID id, @RequestBody
StudyProgramDto studyProgramDto ) {
  try {
    Optional<StudyProgram> found = studyProgramRepository.findById( id );
    if( found.isEmpty() ) {
      // create new entity so that the ID has been set, then transfer the received DTO
      StudyProgram newStudyProgram = new StudyProgram( id );
      modelMapper.map( studyProgramDto, newStudyProgram );
      studyProgramRepository.save(newStudyProgram);
      return new ResponseEntity( HttpStatus.CREATED );
    }
    else {
      // just update
      StudyProgram sp = found.get();
      modelMapper.map( studyProgramDto, sp );
      studyProgramRepository.save( sp );
      return new ResponseEntity( HttpStatus.NO_CONTENT );
    }
  }
  catch( Exception e ) {
    // something bad happened -
    return new ResponseEntity( HttpStatus.UNPROCESSABLE_ENTITY );
  }
}
```

StudyProgramController

- Der Controller wird sozusagen aus Code-Schnipseln von Get One (Test auf ID) und Post (Verarbeitung des Bodies) kombiniert.