Vorlesung "Softwaretechnik"
Entwurf: Architektur

Lutz Prechelt
Freie Universität Berlin, Institut für Informati.

- Archit. = Gesamtstruktur
  - nichtfunkt./funkt. Anforderngn.
  - globale Eigenschaften
- Wiederverwendbare Architekturideen
  - Architekturstile
  - Standard-Architekturen
- Modularisierung
  - Modulbegriff: Schnittstelle, Vertrag, Geheimnis
  - Kriterien für die Aufteilung

Lernziele

- Verstehen, was eine Architektur ist und wozu das wichtig ist
- Grobe Eigenschaften einiger Architekturstile verstehen
- Einige Elemente der Standardarchitektur webbasierter Systeme verstehen
- Verstehen, was eine Modularisierung ist und was eine gute Modularisierung ausmacht

Wo sind wir gerade?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - **Abstraktion**
  - **Wiederverwendung**
  - Automatisierung
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - **Entwurf**
  - Qualitätssicherung
  - Projektmanagement

Wo sind wir?: Entwurf

- Einsicht: Man sollte *vor* dem Kodieren über eine günstige Struktur der Software nachdenken
  - und diese als Koordinationsgrundlage schriftlich festhalten
- Prinzipien:
  - **Trennung von Belangen**
  - ***Architektur***: Globale Struktur festlegen (Grobentwurf), insbes. für das Erreichen der nichtfunktionalen Anforderungen
  - **Modularisierung**: Trennung von Belangen durch Modularisierung, Kombination der Teile durch Schnittstellen (information hiding, Lokalität)
  - ***Wiederverwendung***: Erfinde Architekturen und Entwurfsmuster nicht immer wieder neu
  - **Dokumentation**: Halte sowohl Schnittstellen als auch zu Grunde liegende Entwurfsentscheidungen und deren Begründungen fest

Wo sind wir? (2)

Wo steht man nach der Anforderungsermittlung?

- Man weiß (mehr oder weniger) **was** man bauen will. D.h.:
- Man kennt die funktionalen Anforderungen
  - z.B. beschrieben durch Use Cases etc.
  - und konkretisiert durch das Analysemodell
- Man kennt die nichtfunktionalen Anforderungen
  - wenige oder viele, lasch oder stringent
  - betreffend z.B. Effizienz, Bedienarten, Bedienbarkeit, Robustheit, Sicherheit, Schutz, Erweiterbarkeit, Technologievorgaben, Zuverlässigkeit, Verfügbarkeit, Erlernbarkeit u.a.m.

Wo sind wir? (3)

Was müssen wir als nächstes leisten?

Herausfinden **wie** man es bauen kann/sollte.
D.h.:

1. Entscheiden wie/wo/wodurch die funktionalen Anforderungen (Funktionen) umgesetzt werden
   - z.B. Technologieauswahl, Modulzerlegung, Wiederverwendung
     - Wir klammern Technologieauswahl und Wiederverwendung heute aus
   - Frage 1 ist also: **Wie zerlegt man ein System klug in Teile?**
2. Herausfinden, wie man dabei die nichtfunktionalen Anforderungen alle erfüllen kann
   - nichtfunktionale Anforderungen haben meist globalen Charakter, werden also nicht von nur wenigen Modulen realisiert
   - Frage 2 ist also: **Wie findet man eine Gesamtstruktur mit den gewünschten globalen Eigenschaften?**

Suche nach einer Gesamtstruktur

Freie Universität Berlin

- Das Problem bei der Suche nach einer geeigneten Gesamtstruktur ist folgendes:
  - Die globalen Eigenschaften sind meist emergente Eigenschaften
    - d.h. sie lassen sich nicht den Einzelteilen zuweisen, sondern entstehen erst aus deren Zusammenwirken
    - z.B. Speicherbedarf, Robustheit, Verfügbarkeit, Sicherheit
  - Emergente Eigenschaften sind sehr schwierig im Voraus abzuschätzen
- Muss ich also mehrere Gesamtstrukturen ausprobieren, bis ich eine passende finde?
  - Meist nicht, denn es gibt Erfahrungswerte
  - Die gewünschten Eigenschaften verschiedener Systeme ähneln einander
  - und gewisse Arten von Gesamtstruktur haben sich bewährt

Suche nach einer Gesamtstruktur (2)

- Meist gibt es ein paar dominante nichtfunktionale Eigenschaften
- und eine bekanntermaßen dazu gut passende Gesamtstruktur

- Dann benutzt man solche bekannten Gesamtstrukturen
  - oft "Standardarchitekturen" genannt

- Wenn Sie ein ganz ungewöhnliches, neuartiges System bauen wollen, haben Sie es allerdings evtl. schwer!
  - Deshalb lohnt es sich, Neuartigkeit zu begrenzen (wo das geht)
  - Siehe normales vs. radikales Vorgehen

- Wir üben an einem anschaulichen Beispiel: Häuser →

Dominante Architekturziele bei Häusern: Mobilität, Verteidigbarkeit

Freie Universität Berlin

Mobilität

Verteidigbarkeit

Nachteile jeweils?

Achtung: Das waren nur die Hauptziele.
z.B. Wärmeisolation?, effiz. Verkehr?

Was, wenn ich Mobilität **und** Verteidigbarkeit brauche?

Definition Softwarearchitektur

Freie Universität Berlin

- *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
    - plus 8 weitere Absätze mit Einzelheiten
  - Bass, Clements, Kazman: *"Software Architecture in Practice"* (2nd edition), Addison-Wesley 2003
  - Diese ganze und zwei Dutzend weitere Definitionen siehe https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513807

- Diese Definition betont also die **Module** ("elements"), deren **Schnittstellen** ("visible properties") und die Art, wie diese miteinander in **Beziehungen** stehen ("relationships")

Definition Softwarearchitektur (2)

Freie Universität Berlin

- Andere Definitionen betonen z.B.:
  - die Entscheidungen, die man zur Bildung der Struktur getroffen hat (inkl. Auflösen von Konflikten wie Effizienz ←→ Wartbarkeit)
  - die Erfüllung der funktionalen *und* nichtfunktionalen Anforderungen, die damit erreicht wird
  - die Wiederverwendung von Teilen
  - das dynamische Verhalten des Systems
  - u.a.

unser Fokus

- Es gibt also keine einheitliche Definition für Architektur
- Aber es gibt eine Gemeinsamkeit aller Definitionen:
  Es geht stets um eine globale Betrachtung!
  - Die aber, wenn man sie gründlich macht, bis auf die Detailebene herunterreicht (bei uns: Thema "Modularisierung", 2. Teil)
  - und sich aus vielen Einzelbetrachtungen zusammensetzt
  - Deshalb gibt es z.B. so viele UML-Diagrammarten
    - → verschiedene Sichten und Detailniveaus

Beispiele für standardisierte Ideen für SW-Architekturen

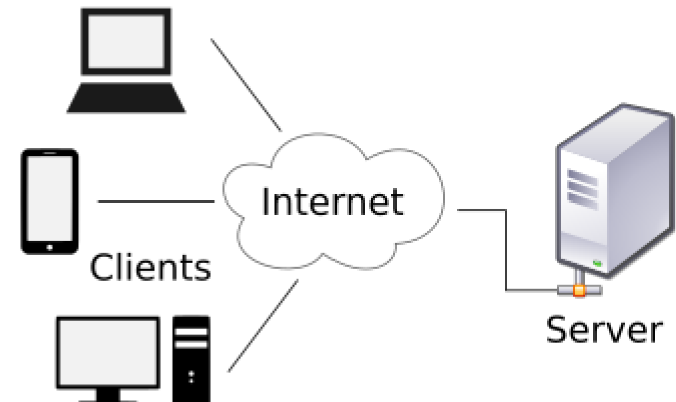
Freie Universität Berlin

- Es gibt für eine Reihe wiederkehrender Mengen nichtfunktionaler Anforderungen etablierte SW-Architekturen oder zumindest grobe Architekturstile, z.B.
  - Klient-/Dienstgeber-Architektur (client/server arch.)
  - Mehrschicht-Architektur (layered arch.)
  - Ereignisgesteuertes System (event-based arch.)
  - Datenflussnetze (pipes-and-filters arch.)
  - Web-Architektur (web-based arch.)
  - (und ein paar andere)
- Reale Systeme *kombinieren* viele der damit gelösten Belange
  ➔ Kombination mehrerer Stile (neben- und ineinander)
  - global (Gesamtsystem)
  - regional (einzelnes Subsystem)

Beispiel 1: Client/Server-Architektur (Klient-/Dienstgeber-Architektur)

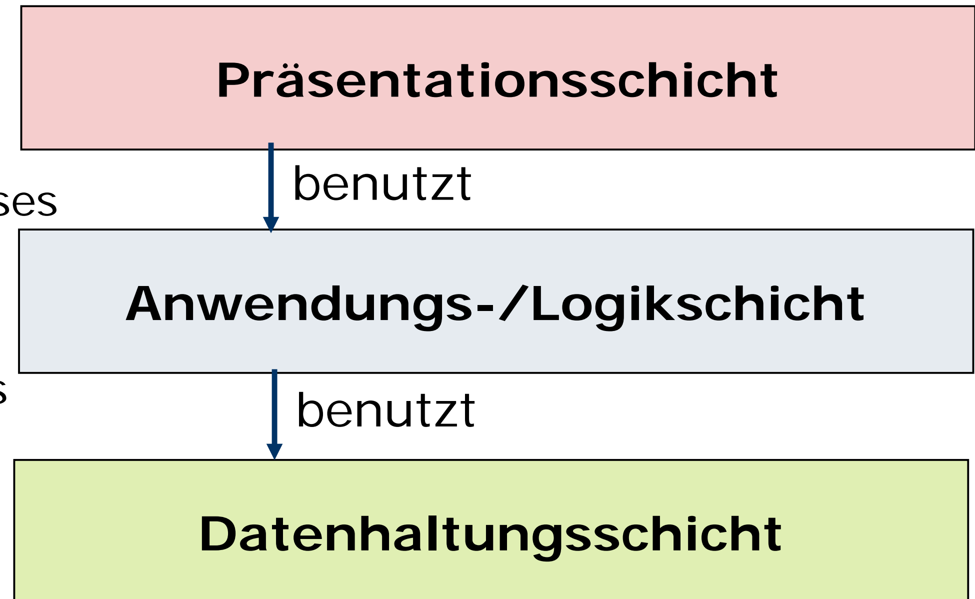
Freie Universität Berlin

- Dominante nichtfunktionale Anforderungen:
  - verteiltes System; gemeinsame Datenhaltung mit evtl. großen Datenmengen; hohe Verfügbarkeit; Interaktivität
- Lösungsidee:
  - gemeinsame Datenhaltung: zentraler Rechner (Server)
    - evtl. mit Redundanz für Verfügbarkeit
  - Interaktivität, Verteilung: zahlreiche Klienten
- Selbstverständlich? Nein, denn:
- Nötige Annahmen:
  - Netze ständig verfügbar
  - Netze leistungsstark genug
  - Klienten haben Netzzugang



Beispiel 2: 3-Schichten-Architektur

- Ein besonders häufiger Fall:
  Die 3-Schichten-Architektur für Informationssysteme
  - Die Präsentationsschicht kapselt die Interaktionen mit Benutzern oder Systemen
    - vor allem Boundary classes
  - Die Logikschicht enthält die Geschäftslogik
    - vor allem Control classes
  - Die Datenschicht kümmert sich um die persistente Speicherung aller Daten
    - vor allem Entity classes



  - Jedes Modul benutzt nur Module seiner Schicht und der Schichten darunter (und liefert Dienste für die höheren Schichten)
    - Höhere Schichten werden von niederen niemals benutzt

- Notation: informell

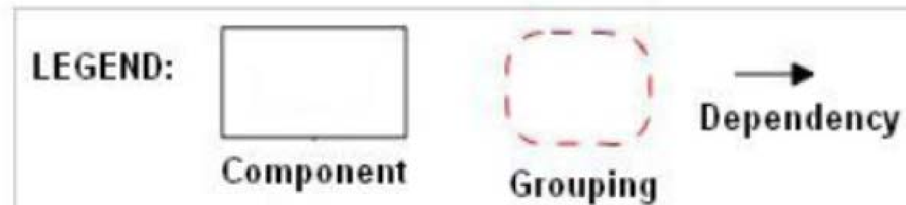
Architektur<u>stil</u> Schichten

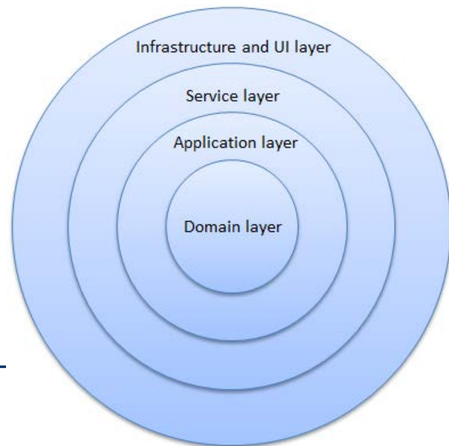
- Idee: Bilde mehrere Abstraktions-schichten, um
  - von technischen Mechanismen (unterste Schicht)
  - zu inhaltlichen Lösungen (oberste Schicht) zu kommen
- Universell nützlich

User Interface
UI Toolkit (XPFE)
User Interface
Browser Engine
Rendering Engine
Gecko
User, Secure, & Engine Persistence
Data Persistence
Necko
Security (NSS/PSM)
Networking
Spider–Monkey
JavaScript Interpreter
Expat
XML Parser
GTK+ Adapter
GTK+ / X11 Libraries
Display Backend

Notation: informell

Schichtenarchitektur des Firefox-Browsers





Diskussion Schichten-Architektur

Vorteile:

- Unnötige Kopplungen zwischen Modulen werden vermieden
- Übersichtliche, verständliche Grobstruktur
- Komplettaustausch ganzer Schichten ist möglich
        - Nutzung: unterste manchmal, oberste seltenst, innere so gut wie nie

Mögliche Nachteile:

- Kann umständlich oder unnatürlich sein
    - Abstraktionen evtl. nur vorhanden, um Schichtung herzustellen
- Kann Laufzeit-ineffizient sein
    - wenn man zu viel abstrahiert hat und zu oft "weiterreichen" muss
        - z.B. im ISO/OSI 7-Schichten-Modell

Für Informationssysteme gilt:
Schichtenarch. ist fast immer günstig

- oder evtl. ports and adapters

Warnung: Das Bild ist nicht die Architektur!

Freie Universität Berlin

- **Achtung, ganz wichtig!**:
  Die Symbole und Linien auf einem Architekturbild sind nicht "die Architektur"!

Sondern nur Andeutungen einer Architektur:

1. Sie repräsentieren nur sehr wenige Aspekte der Architektur
   - hier: die Natur und wichtigsten Kommunikationsbeziehungen der Subsysteme
   - viele andere Aspekte bleiben völlig ungenannt (z.B. Steuerung)
2. Selbst diese Aspekte sind nur sehr ungenau wiedergegeben
   - hier z.B.: Wie oft wird kommuniziert? Welche Datenstrukturen? Welche Kommunikationsmechanismen? Wie ausgelöst? etc.

Architektur und Architektursichten

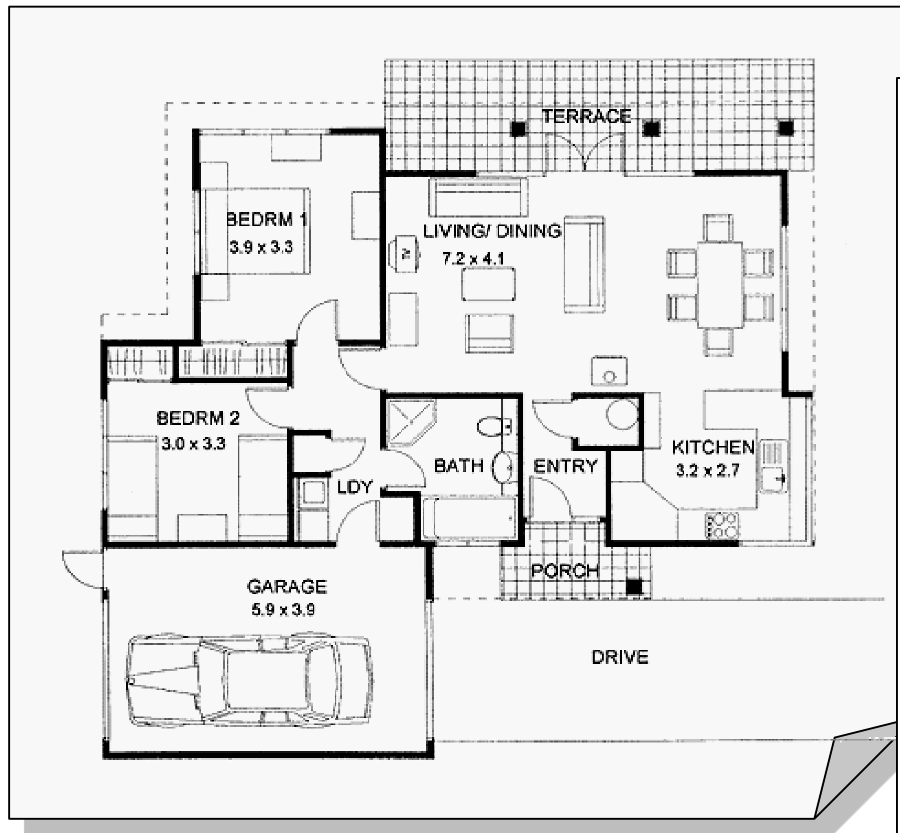
Eine vollständige Architekturbeschreibung ist ein komplexes Modell und besteht aus viel mehr als einem Bild:

1. Man benutzt viele Bilder anstatt nur eines
   - verschiedene Arten von Bild für verschiedene Aspekte
     - z.B. verschiedene UML-Diagrammarten
   - verschiedene Bilder für verschiedene Teilsysteme (auf den feineren Detailniveaus)
     - z.B. viele UML-Teil-Klassendiagramme für das statische Modell

2. Neben den Bildern braucht man zusätzliche Information
   - meist textueller Natur, z.B. Schnittstellendefinitionen
     - z.B. in OCL (Object Constraint Language) oder natürlichsprachlichem Text

Architektur: mehrere Sichten

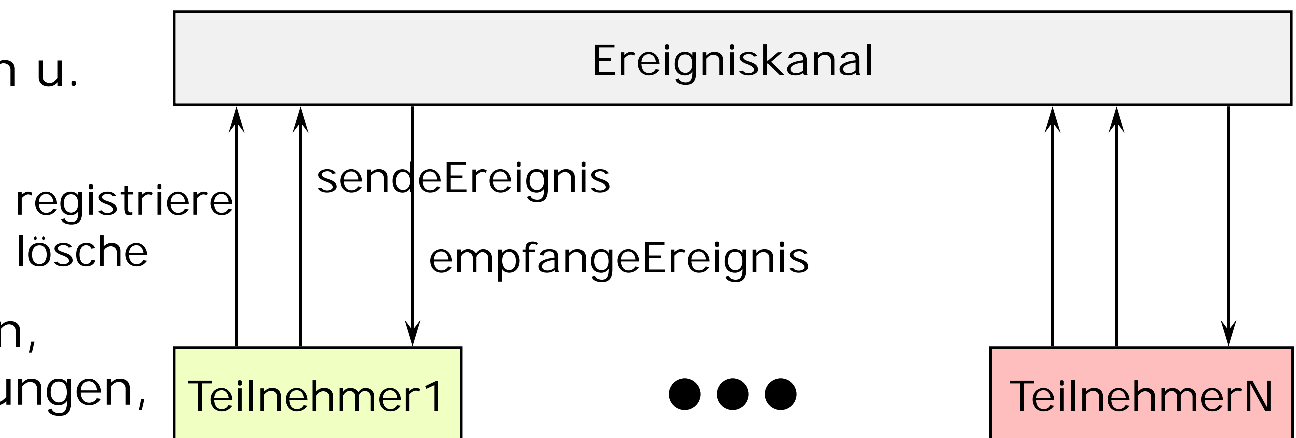
- z.B. Grundriss und Aufriss für ein Haus sind zwei typische Sichten desselben Hauses
  - (hier allerdings zwei verschiedene Häuser)



Architekturstil Ereignissteuerung mit impliziten Aufrufen

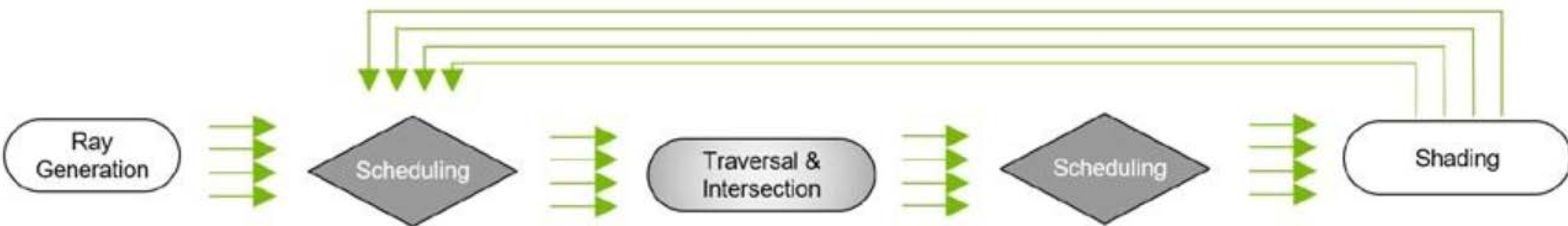
Freie Universität Berlin

- Idee:
  - Verarbeitungseinheiten registrieren sich an zentraler Stelle, um bei bestimmten Ereignissen benachrichtigt zu werden
  - Sie können auch selbst Ereignisse erzeugen
  - Dadurch wird die Systemsteuerung dezentralisiert
- Vorteil:
  - Flexibel und in vielen Fällen änderungsfreundlich
- Nachteil:
  - Verhalten eventuell schwer zu durchschauen u. zu ändern
- Beispiele in
  - GUIs, IoT-Systemen, Fabriksteuerungen, u.v.a.m.



- Idee: Verknüpfe Verarbeitungsschritte (Filters) nur mittels eines Datenflusses (Pipe)
  - Vorteil: Einfach und klar
  - Nachteil: Für die meisten Zwecke zu unflexibel
    - Aber gängig z.B. in der Bioinformatik und der Signalverarbeitung
- Beispiel: Rendering-Pipelines bei Graphikkarten/OpenGL

Notation: informell



Standardarchitektur: Web-basiertes System

Nichtfunktionale Anforderungen:

- (0. alle Anforderungen der Client-/Server-Architektur)
- 1. Interaktion mit Menschen und mit Maschinen
- 2. Internationalisierung (i18n)
- 3. Sicherheit (Vertraulichkeit, Integrität)
- 4. Skalierbarkeit
- 5. semi-dezentrale Entwicklung

➔ Integration sehr vieler Teillösungen

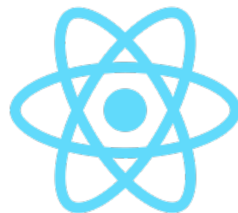
Es folgen ein paar Beispiele:

Anf.: Anforderung *Arch.idee*: Architekturidee
Lsg.teil: Konkrete(re) Teile einer Lösung

Web nichtfkt. Anf.: 1. Interaktion mit Menschen und mit Maschinen

Freie Universität Berlin

- *Anf.*: GUIs + APIs mit wenig Doppelaufwand entwickeln und fortschreiben
  - GUI: Bedienschnittstelle für Menschen (meist Webbrowser)
  - API: Aufrufschnittstelle (über Web) für andere Computer
- *Arch.idee*: Die API auch für den Bau der GUI benutzen
  - Nicht komplette HTML-Seiten an Browser liefern,
  - sondern Javascript, das das HTML dynamisch baut
  - und dafür die API aufruft
- *Lsg.teil*: Javascript-Frameworks wie React oder AngularJS



Web nichtfunktionale Anforderung: 2. Internationalisierung (i18n)

Freie Universität Berlin

- *Anf.*: Die GUI ohne Codeänderung an die Sprachen jedes Landes anpassen können (**`l10n`**: Localization)
- *Arch.idee*: Internationalization (**`i18n`**)
  - Sprachliche Texte *nirgends* in den Code einschließen,
  - sondern immer durch einen Bezeichner ersetzen.
  - Zu jeder Landessprache L ein Lokalisierungspaket vorhalten mit Abbildung Bezeichner → Text in L
    - (es gibt noch mehr Themen: Zeit- und Datumsformate, Zeitzonen, Sprachvarianten, Schreibrichtung u.a.)
  - Im Code eine Bibliothek aufrufen, um den gerade gefragten Text abzurufen.
- *Lsg.teil*: In Java z.B. ResourceBundle + Locale in java.util
  - Tutorial dazu

Web nichtfunktionale Anforderung: 3. Sicherheit (Vertraulichkeit, Integrität)

Freie Universität Berlin

- *Anf.*: Lesezugriffe (Vertraulichkeit) und Schreibzugriffe (Integrität) dürfen nur für Befugte technisch möglich sein

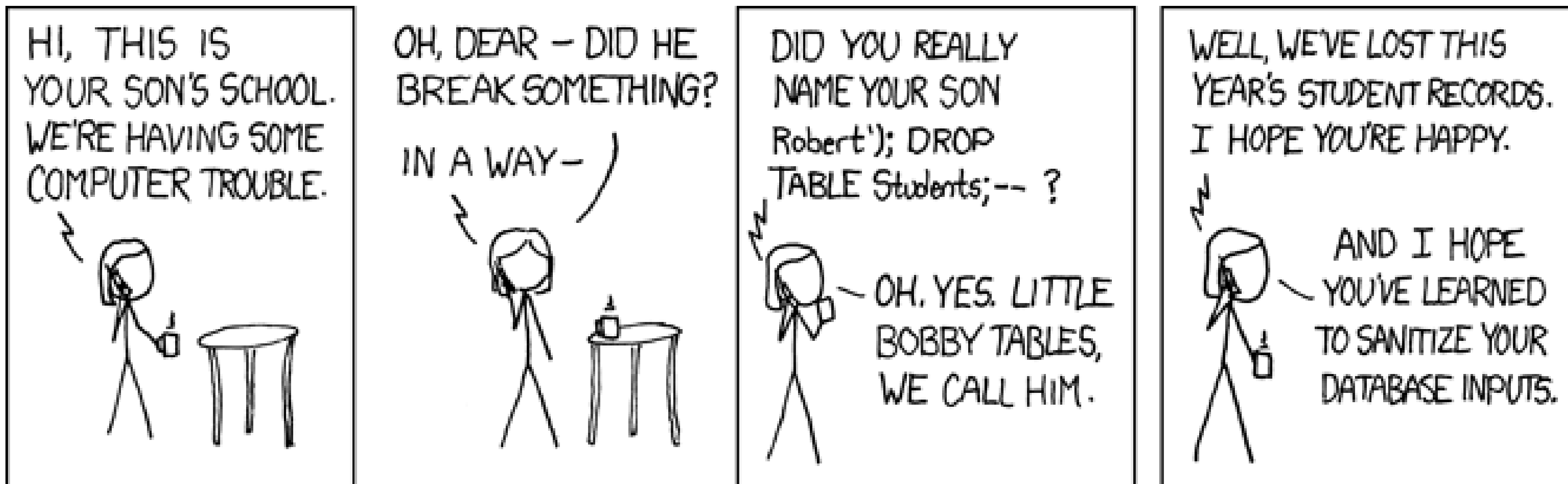
- *Arch.idee*: Verhindere alle Klassen unbefugter Zugriffe z.B.
  - Injektionen (durch Datenprüfung mit Whitelisting),
  - Direktobjektzugriffe (durch spätestmögliches Prüfen der Rechte),
  - Abhören (durch Verschlüsselung)
  - ...

- *Lsg.teil*:
  - Gegen Injektionen gibt es z.T. ausgereifte Bibliotheken
  - Direktobjektzugriffe muss meist die Anwendungslogik selbst abfangen (→ Qualitätssicherung durch Durchsichten)
  - Gegen Abhören hilft Verschlüsselung
    - Es ist leider nicht einfach, Verschlüsselung richtig einzusetzen
  - ...

Sehr problemanfälliges Thema!

Web nichtfkt. Anf.: 3. Sicherheit
Beispiel: SQL-Injektion

- Login: Eingabe von user="prechelt" und password="abc"
- SQL-Datenbankabfrage im Code:
  - SELECT pwd FROM users WHERE (name = '?') (Code)
  - SELECT pwd FROM users WHERE (name = 'prechelt') (Ausführg.)
- Angriff:



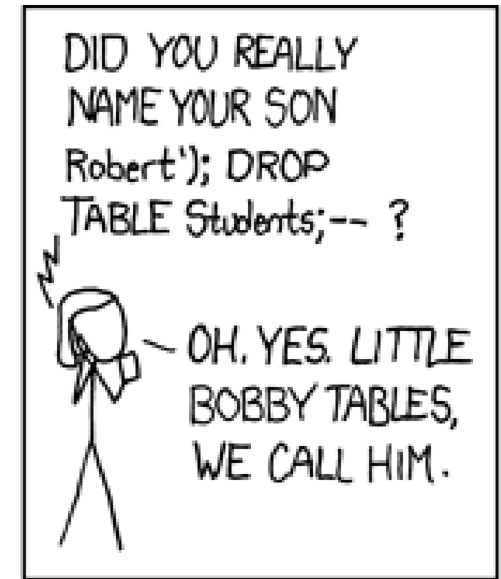
https://xkcd.com/327/

- Was tun?

Web nichtfkt. Anf.: 3. Sicherheit
Beispiel: SQL-Injektion: Lösungen

Freie Universität Berlin

- SELECT pwd FROM users WHERE (name = '?')



- Schlechte Lösung (dieser Fall):
  - ' in den Eingaben verbieten (**Blacklisting**)
    - ' wird in Eingaben evtl. benötigt
    - Reicht u.U. nicht aus: ' auch anders darstellbar
- Bessere Lösung (dieser Fall): prepared statements
  - Lsg.teil: Alle guten SQL-Bibliotheken, z.B. java.sql
    - Auch hier sind die Details wieder weitaus komplizierter

- Bessere Lösung (allgemein): **Whitelisting**
  - Charakterisiere "vernünftige" Eingaben
  - Schreibe Routine, um die zu erkennen
  - Verbiete alles andere

Web nichtfunktionale Anforderung: 4. Skalierbarkeit

Freie Universität Berlin

- *Anf.*: Das System soll durch Zufügen von mehr Servern immer mehr Klienten bedienen können.

- *Arch.ideen* (Formulierung von 12factor.net):
    - (ich erkläre die hier nicht *wirklich*)
  - Load balancing: Ein extra Server weist jede Anfrage einem Server zu
  - VI. Execute the app as one or more stateless processes
  - VIII. Concurrency: Scale out via the process model
  - IV. Treat backing services as attached resources (e.g. DBMS)
  - IX. Disposability: Robustness via fast startup/graceful shutdown

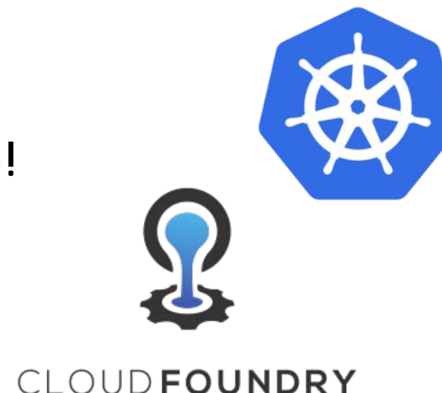
- *Lsg.teil*:
  - Betriebssystem!
  - Zahllose Frameworks, die einzelne Aspekte dieser Ideen unterstützen

Web nichtfunktionale Anforderung: 5. semi-dezentrale Entwicklung

- *Anf.*: Große Webanwendung nicht mit 1 großen Team, sondern mehreren kleinen Teams bauen
  - und die sollen nur hin und wieder kommunizieren müssen.

- *Arch.idee*:
  - Microservices: Baue Subsysteme mit stabilen Schnittstellen als völlig separate Systeme (mit eigener Datenhaltung)
    - Hilft auch für Skalierbarkeit
  - jeder Microservice wird von einem separaten Team gebaut
    - Probleme: Schnittstellen finden! Effizienz!

- *Lsg.teil*:
  - Leistungsstarke Hardware!
  - Zahllose Frameworks, die einzelne Aspekte unterstützen



Standardarchitektur: Web-basiertes System

- Ist **sehr** umfangreich, wir haben das nur zart angetippt.

Standard-Architekturen

- Es gibt weitere Standard-Architekturen und Referenzarchitekturen auf dem Markt
  - in denen z.T. recht detailliert Lösungswege für viele wiederkehrende Probleme vorgezeichnet sind
- Eine solche einzusetzen senkt den Entwurfsaufwand enorm
  - weil große Mengen Expertenwissen darin stecken
- Beispiele:
  - QUASAR (Qualitäts-Software-Architektur) von sd&m
    - generische Architektur für betriebliche Informationssysteme
    - formuliert hauptsächlich als Satz von Prinzipien und Methoden
  - Jakarta EE Architektur
      - vormals Java Enterprise Edition
    - technologie-zentrierte Architektur für Informationssysteme auf Basis zahlreicher eigener Teiltechnologien
  - RM-ODP (Reference Model for Open Distributed Processing)
    - generische Architektur für erweiterbare verteilte Systeme

Genaueres Wissen über Standard-Architekturen

Freie Universität Berlin

- Fast jedes moderne Softwaresystem sollte zumindest weitgehend gemäß einer Standardarchitektur gestaltet sein
  - Normaler Entwurf statt radikaler Entwurf

- Konsequenzen:
  - Befassen Sie sich intensiv mit den Standardarchitekturen in Ihrem späteren Arbeitsbereich
    - Darin steckt sehr viel gutes Know-How ➔ Es gibt viel zu lernen
      - selbst über die simple 3-Schichtenarchitektur
    - Die obigen Darstellungen haben das allenfalls angedeutet
  - Seien Sie sehr argwöhnisch gegen Tendenzen, die benutzte Standardarchitektur zu durchbrechen
    - Kann gelegentlich sinnvoll sein,
    - ist aber sehr oft eine *schlechte* Idee:
    - → Genaue Begründung und Abwägung verlangen!

Zwischen-Zusammenfassung

1. Eine Architektur beschreibt,
   - welche Teile ein System hat,
   - wie diese zusammenspielen
   - und wie dadurch die funktionalen und nichtfunktionalen Anforderungen erfüllt werden können

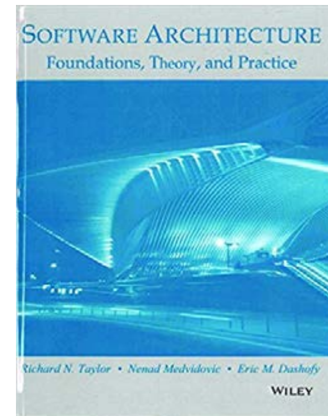
2. Zum Selbsterfinden von Architekturen gibt es wiederkehrende Architekturstile
   - Sollte man ungefähr kennen und verstehen

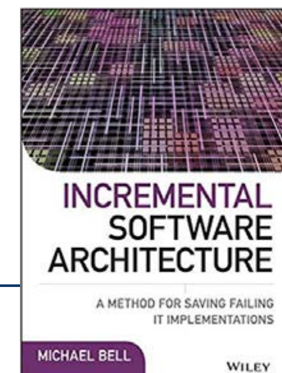
3. Insgesamt stützt man sich aber meist auf eine Standard-Architektur für den gegebenen Anwendungsbereich (Domäne)
   - Mindestens eine davon sollte man sehr genau kennen und verstehen
   - Das Selbsterfinden findet in der Regel nur lokal in einzelnen Komponenten statt

Literatur über Architektur

- Taylor, Medvidovic, Dashofy: *"Software Architecture: Foundations, Theory, and Practice"*, Wiley 2009
  - Umfangreiches Lehrbuch
- Johannes Siedersleben: *"Moderne Software-Architektur"*, dPunkt Verlag 2004
  - über QUASAR
- Michael Bell: *"Incremental Software Architecture: A Method for Saving Failing IT Implementations"*, Wiley 2016
  - moderne Sicht: nimm an, Du kannst es nicht alles in einem Rutsch wissen
- Open Web Application Security Project: OWASP Top 10, https://owasp.org/www-project-top-ten/

OWASP





Nochmal: Wo sind wir gerade?

(nochmal Folie 6)

Was müssen wir als nächstes leisten?

- Herausfinden **wie** man es bauen kann/sollte.
  D.h.:
- Entscheiden wie/wo/wodurch die funktionalen Anforderungen (Funktionen) umgesetzt werden
  - z.B. Technologieauswahl, Modulzerlegung, Wiederverwendung
  - Wir klammern Technologieauswahl und Wiederverwendung aus
  - Frage 1 ist also: **Wie zerlegt man ein System klug in Teile?**
- Herausfinden, wie man dabei die nichtfunktionalen Anforderungen alle erfüllen kann
  - viele nichtfunktionale Anforderungen haben globalen Charakter, werden also nicht von nur wenigen Modulen realisiert
  - Frage 2 ist also: **Wie findet man eine Gesamtstruktur mit den gewünschten globalen Eigenschaften?**

Systemzerlegung

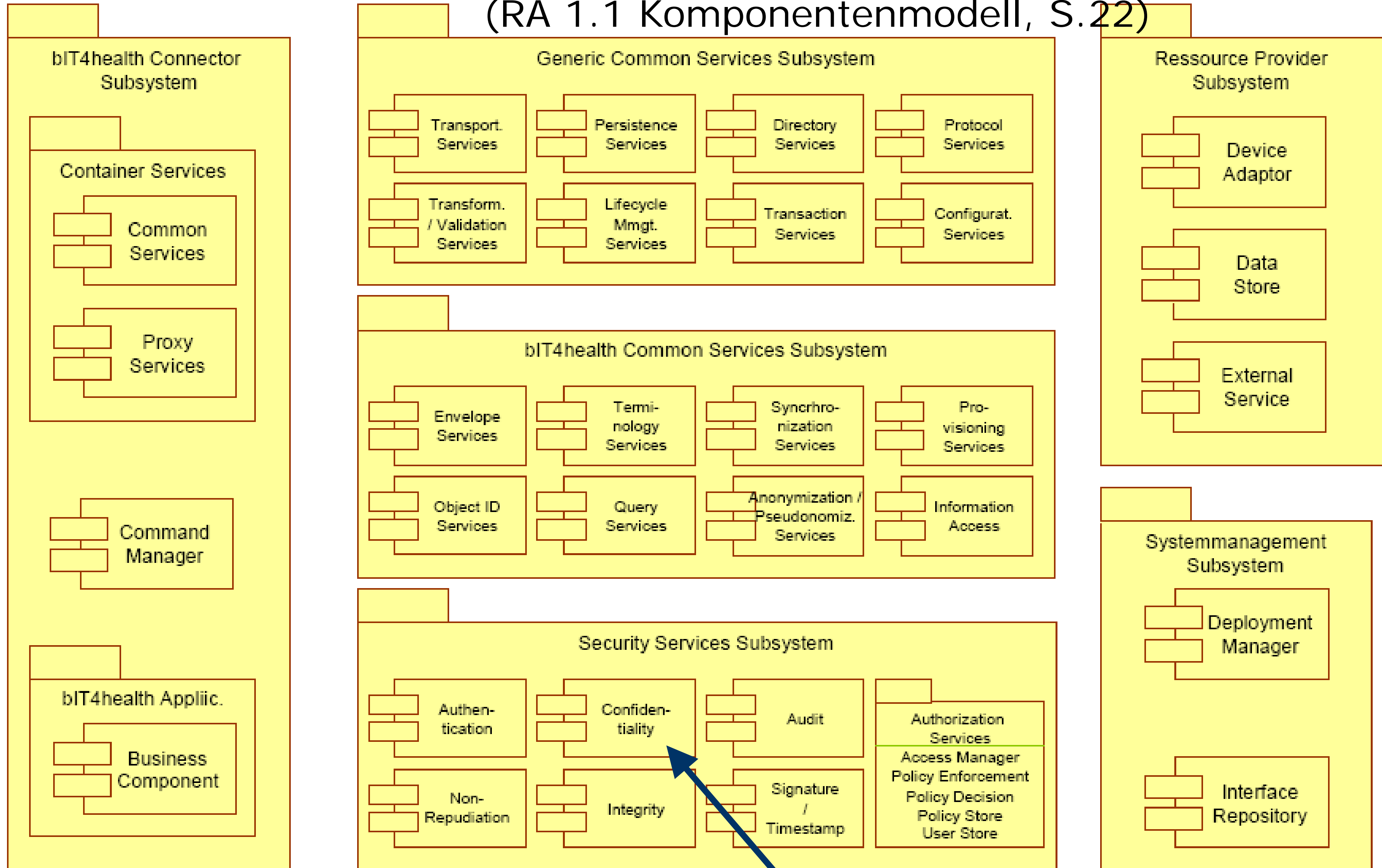
- Ist eine Architektur ausgewählt, muss das System hierarchisch in passende Teile zerlegt werden
  - Diese Teile heißen **Module** (neuerdings auch **Komponenten**)
  - Auf den oberen Ebenen (also für die großen Module) sagt man dazu auch oft **Subsysteme**

- Was gehört zusammen in ein Modul?
  - **Ziel**: Jedes Modul soll möglichst wenig mit den übrigen Modulen zu tun haben (**geringe Kopplung**)
    - sonst ergäben die Module keine richtige "Zerlegung"
  - **Mittel**: Bilde Module nach **Zuständigkeiten** und **Entwurfsentscheidungen**
    - Fasse zusammen, was thematisch zusammengehört (→ verständlich)
    - Fasse zusammen, was bei Korrektur der Entwurfsentscheidung evtl. geändert werden muss (→ änderungsfreundlich)
    - Ganz wichtig: Die **Schnittstelle** muss die Entwurfsentscheidung vor der Außenwelt verbergen (**Geheimnisprinzip**)

Die Hauptsache!

Beispiel: eGK-Subsysteme und -Komponenten

(RA 1.1 Komponentenmodell, S.22)



Notation: UML Komponentendiagramm

Begriff "Modul"

- Ein Modul (in UML: Komponente) ist eine Einheit zusammengehörender Programmelemente
  - z.B. Daten, Datentypen, Klassen, Prozeduren, Funktionen, Prozesse etc.
  - Modulgrenzen sind meist (aber nicht immer) auch Grenzen programmiersprachlicher Konstrukte wie Methode, Klasse, Paket
    - Eine Konstruktgrenze allein macht aber noch lange kein Modul

- Die Elemente gehören thematisch zusammen und weil sie von den selben Entwurfsentscheidungen betroffen sind
  - Das Modul verbirgt diese Entscheidungen vor dem Rest des Systems
  - Man nennt diese Entscheidungen die "**Geheimnisse**" des Moduls
  - Man nennt das Verbergen auch "**Verkapseln**" oder "**information hiding**"

Begriff "Modul" (2)

- Modul ist nach außen nur üb. seine Schnittstelle beschrieben
  - Die Schnittstelle ist die Gesamtheit aller relevanten extern sichtbaren Eigenschaften des Moduls:
  - Die **Funktionen** (bei OO also auch **Klassen**) und ihre genaue garantierte **Semantik**
    - unbedingt also **Voraussetzungen** ("Vorbedingungen", preconditions) und **Effekte** (Wirkungen, "Nachbedingungen", postconditions),
      - in UML z.B. mittels OCL (object constraint language)
    - eventuell auch zugesicherte(!) Eigenschaften wie Laufzeitverhalten, Speicherbedarf, Nebenläufigkeit, Kapazitätsgrenzen, Portabilität usw.

- Das Innere eines Moduls ist in der Regel komplizierter als die Schnittstelle
  - Dadurch reduziert Modularisierung die Systemkomplexität
    - Genau dies ist **der** Trick beim Entwurf
    - Kluge Schnittstellen zu definieren ist eine Kernkompetenz guter Softwareingenieur_innen

Schnittstellen: Entwurf per Vertrag (Design by Contract)

Freie Universität Berlin

Spezifiziere für jede Operation

- **Voraussetzungen** ("Vorbedingungen", preconditions) und **Wirkungen** ("Nachbedingungen", postconditions)

- Diese Semantik definiert einen **Vertrag** zwischen Aufrufer und Modul:

  "Wenn Du, lieber Aufrufer,
  die Voraussetzungen einhältst,
  verspreche ich, die Wirkungen sicherzustellen"

  - Schnittstellen so zu beschreiben heißt auch *Entwurf per Vertrag (design by contract)*
  - Die Verträge werden vor der Implementierung des Moduls festgelegt
    - sie stellen die Modulspezifikation dar

OCL-Beispiel: Methode Tournament.removePlayer

Freie Universität Berlin

context Tournament::removePlayer(p) **pre**:
    isPlayerAccepted(p)

context Tournament::removePlayer(p) **post**:
    not isPlayerAccepted(p)

context Tournament::removePlayer(p) **post**:
    getNumPlayers() = @pre.getNumPlayers() – 1

- Wirkungen ("post") sind oft schwierig auszudrücken
- Aber auch unvollständige OCL-Beschreibungen können nützlich sein!
    - Ist obiges Beispiel vollständig?

Semantik von Schnittstellen und Implementierungen

Freie Universität Berlin

- Meist hat die Implementierung eines Moduls viele Eigenschaften, die nicht in der Schnittstelle beschrieben sind
- Auf solche Eigenschaften darf man sich außerhalb des Moduls nicht verlassen!
  - (sie gehören nicht zum Vertrag, sondern zu den Geheimissen)
  - Sonst hat man einen inkorrekten Entwurf, weil die Eigenschaft nicht wie verlangt verborgen wird
- (Beispiel folgt gleich)
- Leider werden Schnittst. nur selten so genau beschrieben, dass man dieses Prinzip leicht einhalten kann

Beispiel: Semantik von Schnittstellen und Implementierungen

Beispiel:

- Sie rufen nacheinander die Komponenten X und Y und wollen deren Fehlercode speichern, wenn eine davon versagt
  - und auch, *welche* von beiden versagt hat
  - Schnittstelle von Y: definiert Fehlercodes 10 bis 20
  - Schnittstelle von X: definiert nur "non-zero indicates an error"
  - Aber Sie wissen, dass faktisch X dann stets nur 1 liefert
- Sie dürfen aber dieses Wissen nicht ausnutzen
  - und z.B. die Fehlercodes nicht in derselben Variablen ablegen
  - weil eine neue Version von X auch Codes im Bereich 10...20 benutzen könnte

(Falls Sie solches Wissen dennoch mal nutzen: Bitte verkapseln!)

2. Beispiel

Wirkung von Modulen/Modularisierung

- Bei guten Entwürfen:
  - Die Schnittstellen verbergen gerade diejenigen Entwurfsentscheidungen, die sich öfter mal ändern
  - Bei späteren Änderungen muss dann meist nur ein Modul verändert werden, weil dessen Schnittstelle trotz der Änderung gleich bleiben kann
  - **Die Kunst beim Entwerfen liegt darin, solche Schnittstellen zu finden**
- Bei schlechten Entwürfen:
  - Es ist bei Änderungen oft auch die Schnittstelle betroffen und aufrufende Module müssen deshalb mit geändert werden
  - Das passiert vor allem dann, wenn man nicht vorhergesehen hat, was sich ändern könnte

Wirkung von Modulen/Modularisierung (2)

Freie Universität Berlin

- Bei guten Entwürfen:
  Das System ist einfacher zu verstehen
  - Weil man in jeder Situation von den meisten Modulen nur die Schnittstelle verstehen muss
  - aber nicht die dahinter verborgenen Geheimnisse und Details.
  - Modularisierung (d.h. Schnittstellenbildung) reduziert dadurch die effektive Komplexität des Ganzen.

- Bei schlechten Entwürfen:
  Es ist unklar, was alles zur Schnittstelle gehört
    - weil nur Signaturen definiert sind, keine Verträge
  - Deshalb fällt die Komplexitätsreduktion viel geringer aus,
  - weil man das Innere von Modulen betrachten muss,
    um die Schnittstelle zu ermitteln

Verbreitete Missverständnisse über Module

Freie Universität Berlin

- "Ein anständiges Modul ist immer 100-300 Zeilen lang"
  - Unsinn. Ein Modul kann sehr klein sein (1 Zeile) oder sehr groß (1 Million Zeilen – dann enthält es aber sicherlich Teilmodule)
- "Jede Klasse ist ein Modul"
  - Nicht immer. Und: Viele Methoden sind selbst auch Module (verbergen ein Geheimnis)
- "Ein Java interface ist eine Modulschnittstelle"
  - Unsinn. Ein Java interface legt nur die *Signaturen* von Operationen eines Moduls fest.
    - Es fehlt eine Beschreibung der Bedeutung der Operationen
    - Es fehlt eine Beschreibung der nichtfunktionalen Eigenschaften
- "Module sind wiederverwendbar"
  - Meist sind sie es nicht, sondern gemacht nur für ein bestimmtes System. Sie kapseln lediglich eine Entwurfsentscheidung ein.

Modul: männlich oder sächlich?

Das Modul (Plural: Die Module): [englisch: module]

- Technik:
  Baugruppe
  - etymologisch verwandt mit 'Modell'

Der Modul (Plural: Die Moduln): [englisch: modulus]

- Mathematik:
  Teilungsrest (Zahlen),
  eine Klasse algebraischer Strukturen
- Physik, Ingenieurwesen:
  Dimensionslose Maßzahl, ...
  - etymologisch enger verwandt mit 'Maß' (gr. metron, lat. modus)

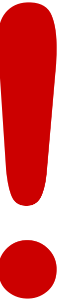
In der Softwaretechnik also immer: Das Modul, die Module

Quelle z.B. Wiktionary [9]

Wie findet man eine gute Modularisierung?

Freie Universität Berlin

- Es gibt nirgends eine wirkliche Anleitung, wie man das anstellt
- Sondern nur bekannte Kriterien zur Beurteilung des Resultats:
  1. Ein Modul hat eine gut verständliche Zuständigkeit
  2. Module sind nicht größer als nötig
  3. Module sind meistens viel komplexer als ihre Schnittstellen
  4. Die Entscheidungen, die sich vielleicht ändern könnten, sind verborgen



- Was ist die große Schwäche dieser Kriterien?
  - Was sich ändern könnte ist oft schwer zu erkennen!
- Deshalb studieren wir das Problem in der nächsten Stunde an einem Fallbeispiel
  - KWIC – Keyword in Context

Zusammenfassung

Freie Universität Berlin

- Eine Architektur beschreibt,
  - welche **Teile** ein System hat,
  - wie diese **zusammenspielen**
  - und wie dadurch die **funktionalen** und **nichtfunktionalen Anforderungen** erfüllt werden können
- Wichtigster Aspekt einer Architektur ist die Modularisierung des Systems (Zerlegung in Module)
  - Ein Modul wird beschrieben über seine **Schnittstelle**
  - Die Schnittstelle verbirgt ein oder mehrere **Geheimnisse** des Moduls
  - Die Geheimnisse spiegeln **Entwurfsentscheidungen** wider, die sich ändern könnten
- Ein gut modularisiertes System ist einfach zu verstehen und Änderungen beschränken sich meist auf ein Modul

Danke!