Programmierkonzepte in den Naturwissenschaften

22. Softwarearchitektur

PD Dr. Till Biskup
Physikalische Chemie und Didaktik der Chemie
Universität des Saarlandes
Sommersemester 2020

Zentrale Aspekte

- Software ist mehr als die Summe der Einzelteile.
  Ihr wechselweises Zusammenspiel ist entscheidend.
- Softwarearchitektur schlägt die Brücke vom sauberen Code einzelner Module zur eigentlichen Anwendung.
- Intuitive Lösungen widersprechen oft dem großen Ziel flexibler, erweiterbarer und wiederverwendbarer Software.
- Gute Architektur fokussiert auf die Funktionalität, nicht auf konkrete Umsetzungen.
- Ein gutes (abstraktes) Modell des Problemraums bildet den Kern einer guten Softwarearchitektur.

Standortbestimmung

Grobgliederung der Vorlesung „Programmierkonzepte“

1. Motivation
2. Infrastruktur
3. Sauberer Code
4. **Softwarearchitektur**
5. Datenauswertung in den Naturwissenschaften

- ☛ Softwarearchitektur vermittelt von der Abstraktionsebene zwischen Code und Anforderungen an das Programm.
- ☛ Softwarearchitektur ist Voraussetzung für die Datenauswertung, wird aber in der Praxis viel zu selten berücksichtigt.

Übersicht

Motivation und Begriffsklärung

Aspekte von Softwarearchitektur

Symptome schlechter und Prinzipien guter Softwarearchitektur

Im Zentrum: ein Modell der komplexen Realität

Ausgangspunkt

Software ist intrinsisch komplex

> *There are many things that make software development complex. But the heart of this complexity is the essential intricacy of the problem domain itself.*
> *If you're trying to add automation to complicated human enterprise, then your software cannot dodge this complexity—all it can do is control it.*
>
> – Martin Fowler

> *In case you haven't realized it, building computer systems is hard. As the complexity of the system gets greater, the task of building the software gets exponentially harder.*
>
> – Martin Fowler

Vorwort in: Eric Evans: Domain Driven Design, Addison-Wesley, Upper Saddle River 2004
Vorwort in: Patterns of Enterprise Application Architecture, Addison-Wesley, Boston 2003

Anforderungen an Software

Flexibel, wiederverwendbar, wartbar – eine Frage der Ökonomie

- flexibel
  - Anforderungen an Software ändern sich grundsätzlich.
  - Anforderungen entwickeln sich parallel zum Verständnis.
- wiederverwendbar
  - Baukasten: Viele elementare Operationen werden immer wieder in unterschiedlichem Kontext benötigt.
  - Modularität und saubere Schnittstellen sind Trumpf.
- wartbar
  - Software sollte ihren Erstentwickler überdauern können.
  - stellt Ansprüche an Softwarequalität und Wissenstransfer

☛ Gleiche Anforderungen wie an „Sauberen Code“ – nur auf einer höheren Abstraktionsebene

> *It doesn't take a huge amount of knowledge and skill to get a program working. [...] It works because getting something to work—once—isn't that hard.*
>
> *Getting it right is another matter entirely.*
> *Getting software right is* hard*.*
> *It takes knowledge and skills that most young programmers haven't yet acquired. It requires thought and insight that most programmers don't take the time to develop.*
> *It requires a level of discipline and dedication that most programmers never dreamed they'd need.*
> *Mostly, it takes a passion for the craft and the desire to be a professional.*
>
> – Robert C. Martin

Robert C. Martin: Clean Architecture. Prentice Hall, Boston, 2018, S. 1

Softwarearchitektur

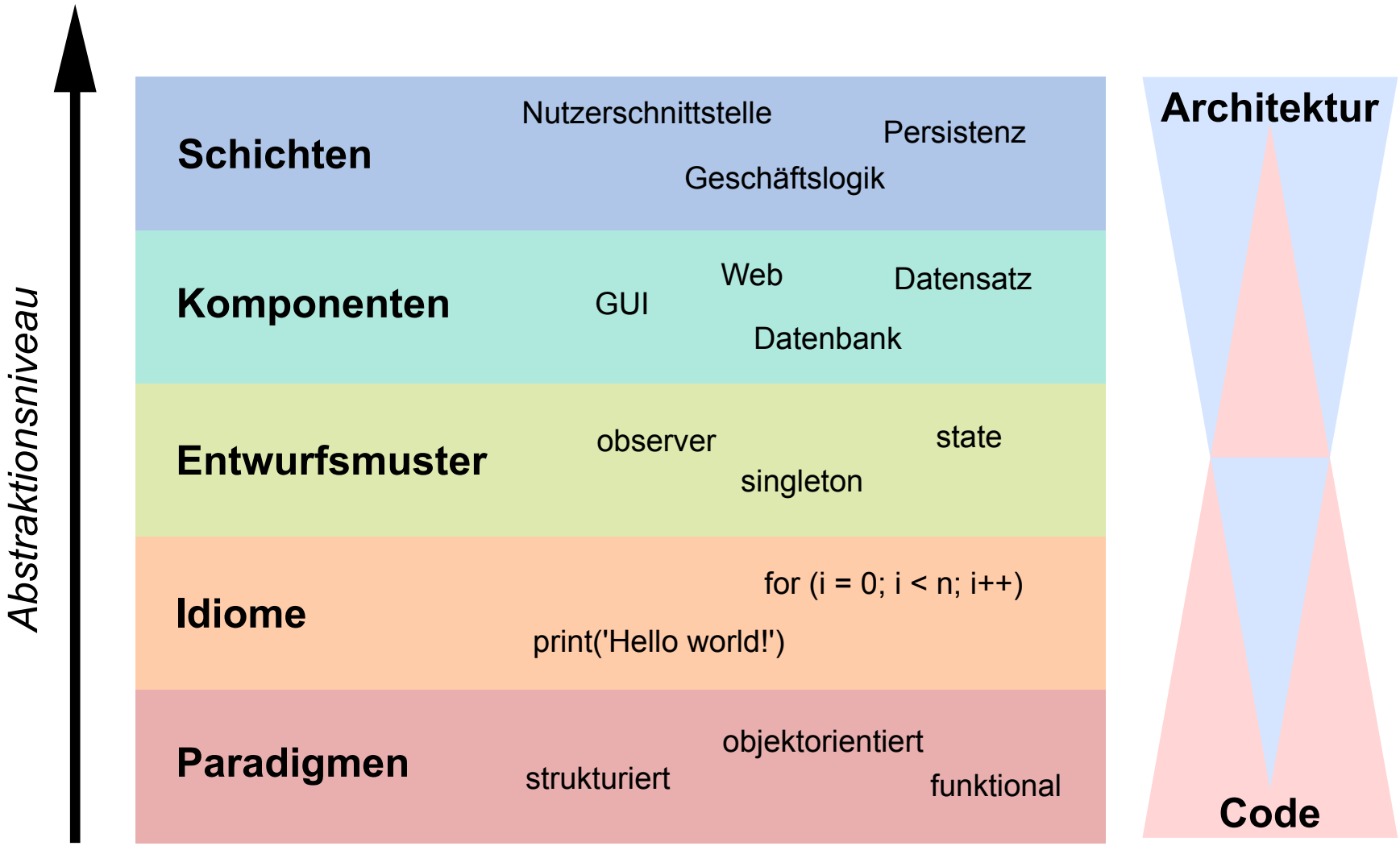
Gestalt eines Systems, die ihm von seinen Entwicklern gegeben wird:
Unterteilung des Systems in Komponenten, ihre Anordnung,
und die Art ihrer Interaktion miteinander.

- Fokus auf dem (Gesamt-)System
  - höhere Abstraktionsebene als bislang
- Entwickler sind die Urheber
  - Architektur wird nicht von außen aufgezwungen.
- Interaktion der Komponenten
  - Definition (möglichst) stabiler Schnittstellen
  - „Gesetz von Conway“: Softwarearchitektur ist ein Abbild der real existierenden Kommunikationsstrukturen

Robert C. Martin: Clean Architecture. Prentice Hall, Boston, 2018, S. 136

Zusammenhang zwischen Code und Architektur

Eine Frage der Abstraktionsebene



Übersicht

Grundsätzliche Strategien

Modularität und möglichst späte Festlegung

> *The strategy behind [software architecture] is to leave as many options open as possible, for as long as possible.*
>
> – Robert C. Martin

- starke Kohäsion (*high/strong cohesion*)
  - jede Einheit hat eine Aufgabe
  - alle Teile der Einheit dienen dem Zweck, diese Aufgabe zu erfüllen
- lose Kopplung (*low/loose coupling*)
  - wenige Abhängigkeiten von Einheiten untereinander
  - erleichtert die (separate) Wiederverwendbarkeit

Robert C. Martin: Clean Architecture. Prentice Hall, Boston, 2018, S. 136

Grundsätzliche Strategien

Weitestgehende Entkopplung auf allen Ebenen

- Trennung der Zuständigkeiten (*separation of concerns*)
  - grundlegendes Prinzip auf vielen Ebenen
  - Bsp.: Funktionen sollten genau eine Aufgabe erfüllen.
  - Kapselung und damit „Verstecken“ von Information
  - Schichten eines komplexeren Programms (s.u.)
  - Führt zu Modularität und Austauschbarkeit von Codeteilen.
- „Gesetz von Demeter“
  - Objekte sollten nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren.
  - Verringert die Abhängigkeiten und erhöht die Wartbarkeit.
  - als Regel für objektorientierte Programmierung entwickelt

☛ Änderungen eines Aspekts sollten lokal klar begrenzte Auswirkungen auf den Code des Gesamtprojekts haben.

Modularität führt zu mehreren Schichten

Schichten sollten so unabhängig wie möglich voneinander sein.

- normalerweise (mindestens) drei Schichten
  - Präsentation
  - Verarbeitung
  - Daten
- je nach Komplexitätsgrad weitere Schichten
  - Zwischenschichten zwischen den oben genannten
  - Persistenzschicht für die Daten (Speicherung)
- Die Schichten sollten so unabhängig wie möglich sein.
  - hilft bei Austauschbarkeit und Wiederverwendbarkeit
  - Änderungen sollten immer nur eine Schicht betreffen.

☛ Kontrollfluss und Abhängigkeiten zeigen nicht zwingend in die gleiche Richtung.

Fokus auf Funktionalität

Entscheidend sind Funktion und Verhalten, nicht Implementierung

> *A good architect maximises the number of decisions not made.*
>
> – Robert C. Martin

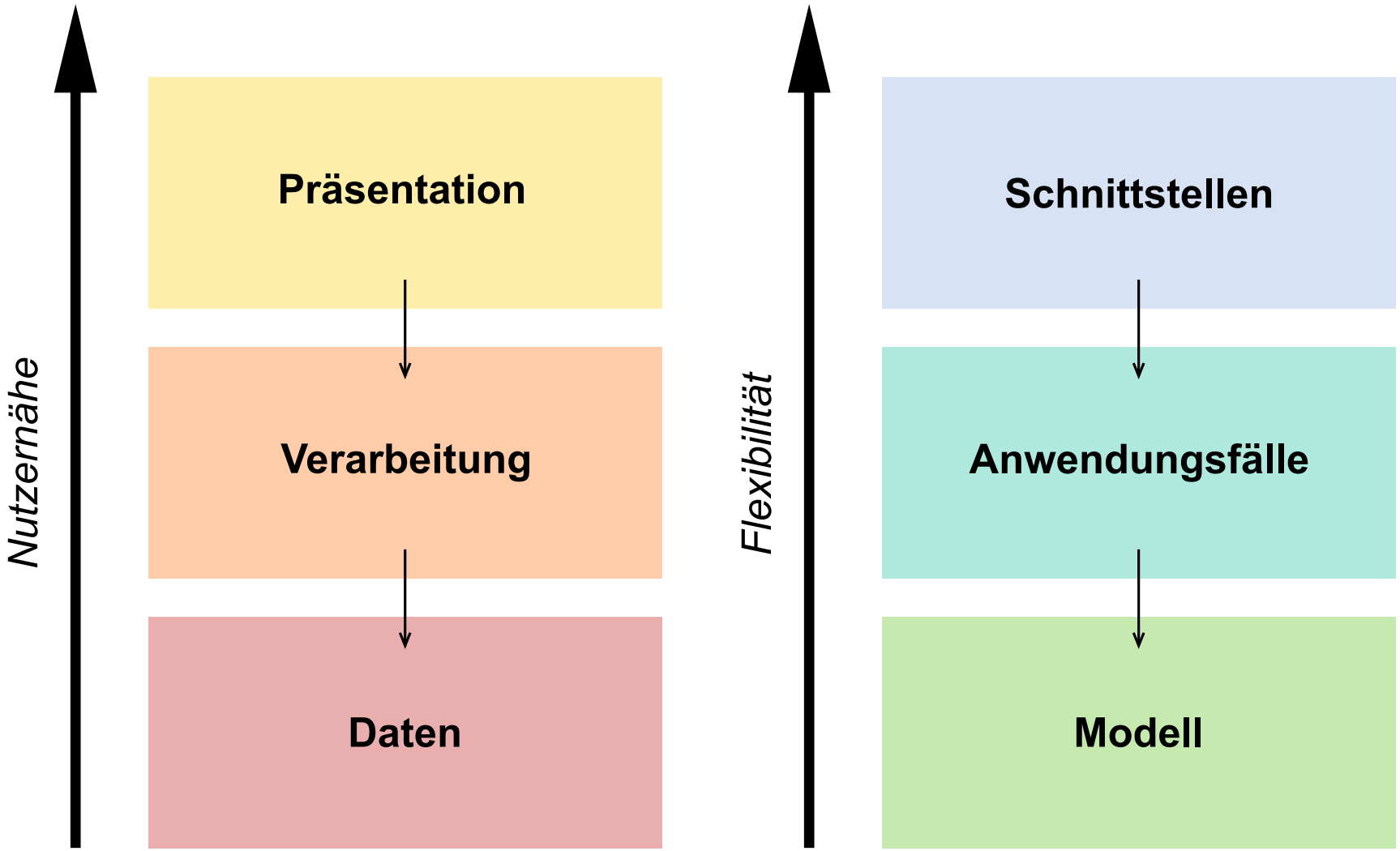
- Die Verarbeitungsschicht sollte im Zentrum stehen.
  - Weder Präsentation noch Datenschicht sind zentral.
  - Präsentation und Datenschicht sollten austauschbar sein.
- Abhängigkeiten und Kontrollfluss trennen
  - Eine Präsentation sollte nicht (direkt oder indirekt) von der Datenschicht abhängen.
  - Trennung erlaubt hochgradige Modularität.

☛ Dieser Perspektivenwechsel führt mitunter zu überraschenden Ergebnissen – und eleganter Architektur.

Robert C. Martin: Clean Architecture. Prentice Hall, Boston, 2018, S. 142

Modularität führt zu mehreren Schichten

Zwei (von vielen) Möglichkeiten für Schichten einer Software



Übersicht

Symptome schlechter Softwarearchitektur

„*Architectural smells*“

> *If it stinks, change it.*
>
> – Grandma Beck

Beispiele für *Architectural Smells* nach Robert C. Martin

- Starrheit (*rigidity*)
- Zerbrechlichkeit (*fragility*)
- Unbeweglichkeit (*immobility*)
- Zähigkeit (*viscosity*)
- unnötige Komplexität (*needless complexity*)
- unnötige Wiederholung (*needless repetition*)
- Intransparenz (*opacity*)

zitiert in: Fowler: Refactoring, Addison-Wesley, Boston 1999, S. 75
Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 88

Symptome schlechter Softwarearchitektur

„*Architectural smells*“

- Starrheit (*rigidity*)
  - Änderungen erzwingen Änderungen an anderen Teilen des Systems
  - fehlende Orthogonalität: „Alles hängt mit allem zusammen.“
- Zerbrechlichkeit (*fragility*)
  - Änderungen führen zu Fehlern in anderen Teilen des Systems
  - starke Kopplung der einzelnen Komponenten
- Unbeweglichkeit (*immobility*)
  - System ist schwer in wiederverwendbare Komponenten aufteilbar
  - fehlende Modularität – System meist „organisch“ gewachsen
- Zähigkeit (*viscosity*)
  - Dinge richtig zu machen ist schwerer, als sie falsch zu machen.
  - zwei Varianten: Software und Entwicklungsumgebung

Symptome schlechter Softwarearchitektur

„*Architectural smells*“

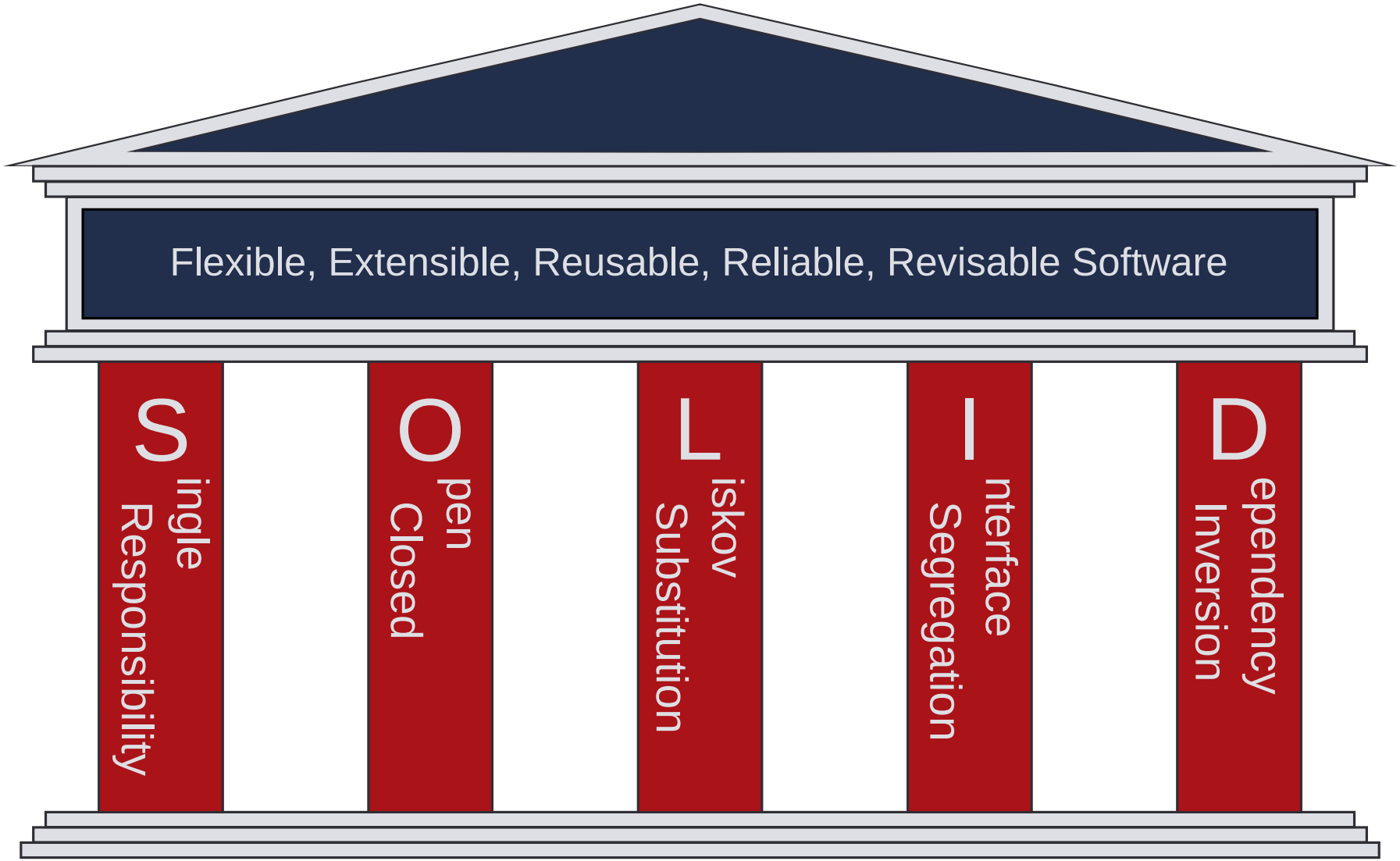
- unnötige Komplexität (*needless complexity*)
  - Entwurf enthält Komponenten ohne direkten Nutzen
  - YAGNI – *You ain't gonna need it!*
- unnötige Wiederholung (*needless repetition*)
  - Strukturen wiederholen sich, statt abstrahiert einmal vorzuliegen.
  - DRY – *Don't repeat yourself!*
- Intransparenz (*opacity*)
  - schwer les- und verstehbar, drückt seine Absicht nicht klar aus
  - Lesbarkeit ist auch auf Architekturebene essentiell.

These

Das Auftreten der Symptome ist normal.
Entscheidend ist, etwas dagegen zu unternehmen.

Fünf Prinzipien guter Softwarearchitektur

Ein erster Überblick



Fünf Prinzipien guter Softwarearchitektur

Ein erster Überblick

- Single Responsibility
  - Ein Modul sollte nur eine Verantwortlichkeit haben.
- Open Closed
  - Software-Einheiten sollten offen für Erweiterung, aber verschlossen gegenüber Abänderung sein.
- Liskov Substitution
  - Subtypen müssen durch ihre Basistypen ersetzbar sein.
- Interface Segregation
  - Module sollten von nichts abhängen, das sie nicht nutzen.
- Dependency Inversion
  - Abstraktionen sollten nicht von Details abhängen. Umgekehrt sollten Details auf Abstraktionen aufbauen.

Das Ergebnis jahrzehntelanger Erfahrung

Voraussetzung: objektorientierte Programmierung

- in der Praxis bewährt
  - Ergebnis jahrzehntelanger Erfahrung von Profis
  - nicht nur die Idee eines Autors
  - Die Prinzipien sind älter als ihr Name.
- hier objektorientiert vorgestellt
  - Objektorientierung erleichtert die Umsetzung.
  - Die Prinzipien sind darüber hinaus gültig.
- Prinzipien sind erst einmal abstrakt.
  - Anwendung auf die konkrete Situation erfordert Bewusstsein, Kenntnis und viel Erfahrung.

☛ Diese Prinzipien sind kein Heilsversprechen,
aber hilfreich für qualitativ hochwertige Programme.

Prinzipien sind kein Selbstzweck

Voreilige Verwendung führt zu unnötig komplexem Code.

> *It is a mistake to unconditionally conform to a principle just because it is a principle.*
>
> – Robert C. Martin

- Fokus auf funktionierender, existierender Software
  - bestmögliche Lösung für bestehende Anforderungen
  - Nur existierende Software kann sich bewähren...
- Prinzipien erst anwenden, wenn notwendig
  - Voreilige Verwendung führt zu unnötig komplexem Code.
  - Kenntnis der Prinzipien und Vertrautheit mit ihrer Umsetzung sind zwingende Voraussetzung.

Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 86

Übersicht

Im Zentrum: ein Modell der komplexen Realität

Kerngedanke des *Domain-Driven Design*

Domain-Driven Design

- Modell der *komplexen* Realität im Zentrum der Betrachtung

Modell

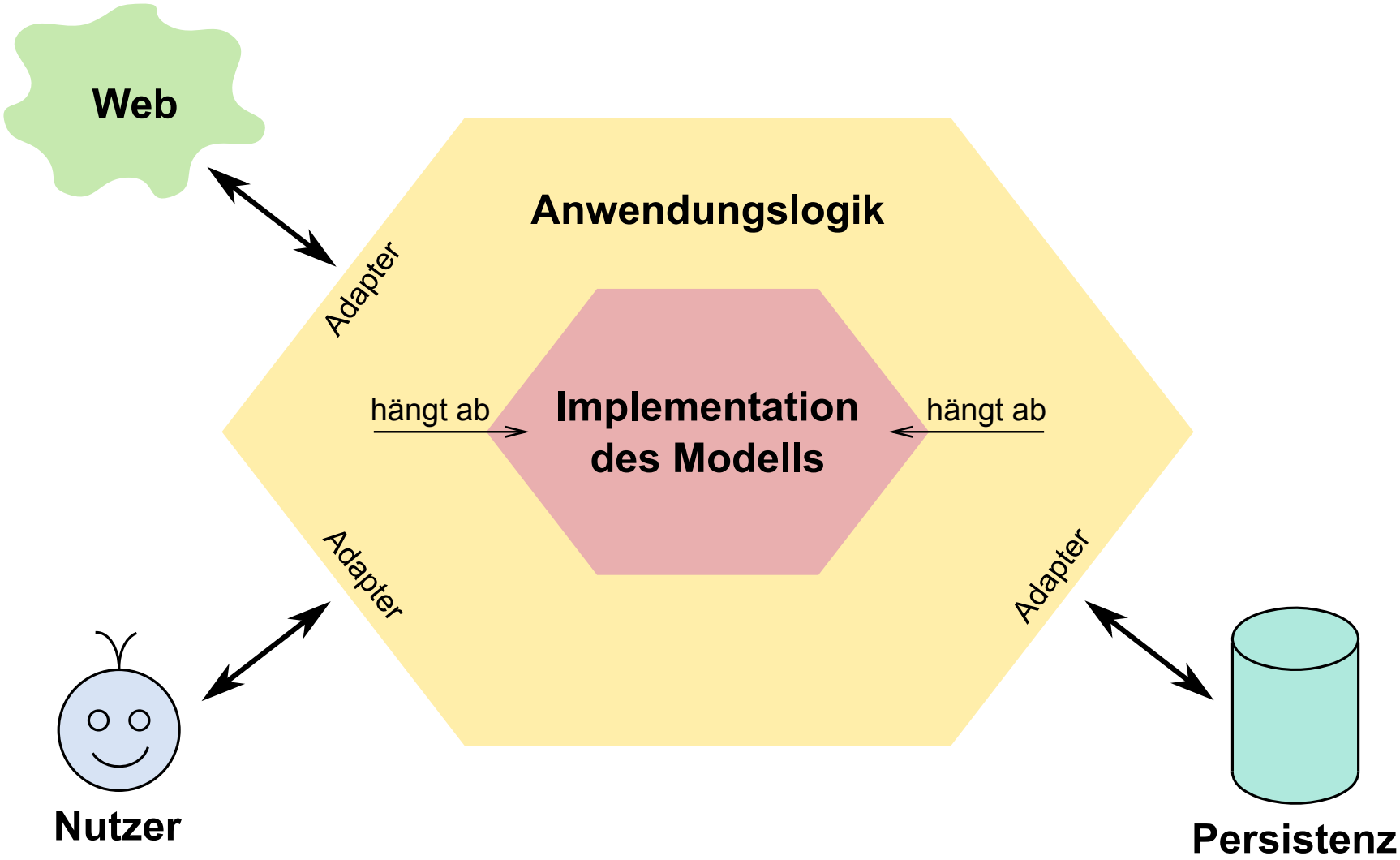
Sprachliche Formulierung des Problemraums und seiner entscheidenden Zusammenhänge und Abläufe auf hohem Abstraktionsniveau;
Summe der „Geschäftsregeln“

Kern der Anwendung

Implementation des Modells in Software, d.h. Abbildung auf Code, zumeist in Form abstrakter Klassen

Im Zentrum: ein Modell der komplexen Realität

Das Modell ist unabhängig von allen anderen Schichten



Im Zentrum: ein Modell der komplexen Realität

Das Modell ist unabhängig von allen anderen Schichten

> *The key to controlling complexity is a good domain model,*
> *a model that goes beyond a surface vision of a domain*
> *by introducing an underlying structure, which gives*
> *the software developers the leverage they need.*
> *A good domain model can be incredibly valuable,*
> *but it's not something that's easy to make.*
> *Few people can do it well, and it's very hard to teach.*
>
> – Martin Fowler

- Ein gutes Modell erfordert detaillierte Kenntnis der Problemstellung und die Fähigkeit zur Abstraktion.
- Abstraktionen müssen treffende Namen tragen.
  Das erfordert gute beschreibende Fähigkeiten.

Vorwort in: Eric Evans: Domain Driven Design, Addison-Wesley, Upper Saddle River 2004

Drei Aspekte von „Domain Driven Design“

Wege zu einem Modell

- Modell und Kern des Entwurfs formen sich gegenseitig.
  - Relevanz des Modells durch Nähe zur Implementierung
  - Verständnis des Modells hilft bei Interpretation des Codes
- Das Modell ist Grundlage einer gemeinsamen Sprache.
  - erleichtert Kommunikation zwischen Entwicklern der Software und Experten der Problemstellung
  - Sprache zur Verfeinerung des Modells nutzbar
- Das Modell ist kondensierte Kenntnis.
  - Konsens zwischen Entwicklern und Experten
  - ermöglicht frühzeitiges Feedback zwischen Programm- und Modellentwicklung

☛ Das Modell bildet den innersten Kern der Software.

Eric Evans: Domain Driven Design, Addison-Wesley, Upper Saddle River 2004, S. 3f.

Zentrale Aspekte

- Software ist mehr als die Summe der Einzelteile.
  Ihr wechselweises Zusammenspiel ist entscheidend.
- Softwarearchitektur schlägt die Brücke vom sauberen Code einzelner Module zur eigentlichen Anwendung.
- Intuitive Lösungen widersprechen oft dem großen Ziel flexibler, erweiterbarer und wiederverwendbarer Software.
- Gute Architektur fokussiert auf die Funktionalität, nicht auf konkrete Umsetzungen.
- Ein gutes (abstraktes) Modell des Problemraums bildet den Kern einer guten Softwarearchitektur.