

Programmierkonzepte in der Physikalischen Chemie

23. Single-Responsibility-Prinzip

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2018/19



Zentrale Aspekte

- Ein Modul sollte nur Verantwortung gegenüber genau einem Akteur haben.
- Jede Verantwortlichkeit ist eine (potentielle) Quelle für Veränderungen.
- Verantwortlichkeiten richtig zu trennen, ist ein zentraler Aspekt jeglicher Software-Architektur.
- Trennung der Verantwortlichkeiten ist nur dann wichtig, wenn unabhängige Änderungen real auftreten.
- Eines der einfachsten Prinzipien für Software-Architektur – und eines der am schwersten korrekt umzusetzenden.

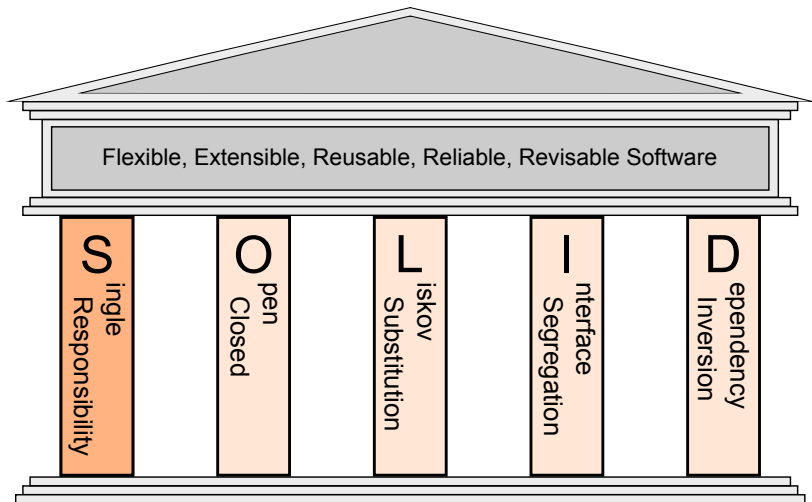
Das Single-Responsibility-Prinzip

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

Das Single-Responsibility-Prinzip

Übersicht über die fünf Prinzipien



“ *A module should be responsible to one, and only one, actor.*

– Robert C. Martin

- ▶ Kontext: Kohäsion
 - Elemente eines Moduls gehören funktional zusammen
- ▶ hier leicht anderer Blickwinkel
 - Welche Kräfte bringen ein Modul (bzw. eine Klasse) dazu, sich zu verändern?
- ▶ (historisch) alternative Formulierung:
 - „A class should have only one reason to change.“

“ *organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.*

– Melvin E. Conway

- ▶ Organisationen bestehen aus unterschiedlichen Einheiten.
 - Arbeitsteilung ist Voraussetzung für Effizienz
- ▶ Jede Einheit hat eigene Verantwortlichkeiten.
 - erfolgreiche Verantwortung setzt Abgrenzung voraus
- ▶ Jede Verantwortlichkeit ist eine Quelle für Veränderungen.
 - „Grund für Veränderung“ ist ein gutes und greifbares Maß.

Was ist eine Verantwortlichkeit?

Eine kontextspezifische Antwort



- ▶ Im Kontext des SRP: ein Grund für Veränderung
 - Wenn man sich mehr als einen Grund vorstellen kann, dann hat das Modul mehr als eine Verantwortlichkeit.
- ▶ Warum sollten Veränderungen gekapselt werden?
 - Voraussetzung für Modularität
 - Veränderungen können sich gegenseitig stören.
 - Verringert die Auswirkungen bei kompilierten Sprachen: Weniger Module müssen neu kompiliert werden.
- ▶ Problem mit Verantwortlichkeiten in Software
 - Wir sind gewohnt, Verantwortlichkeiten zu bündeln.
 - In der Software laufen alle Verantwortlichkeiten zusammen.
 - führt oft zu Modulen mit (zu) vielen Verantwortlichkeiten

Beispiele für seinen Einsatz

Der „Klassiker“: die Gehaltsabrechnung



Finanzen

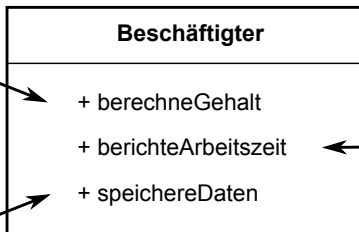


CFO

Technik



CTO



Personal



COO

Beispiele für seinen Einsatz

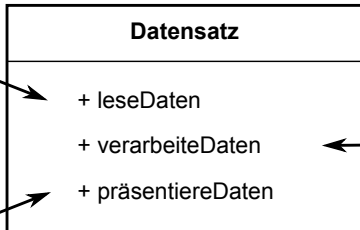
Eine Übertragung in die Wissenschaft: der Datensatz



Gerätehersteller



Betreuer



Student

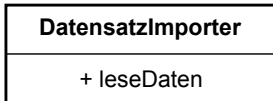


Beispiele für seinen Einsatz

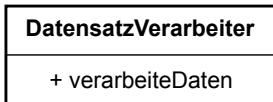
Drei Lösungsmöglichkeiten (1): vollständige Separation in Klassen



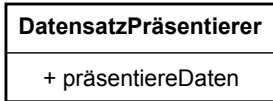
Gerätehersteller



Student

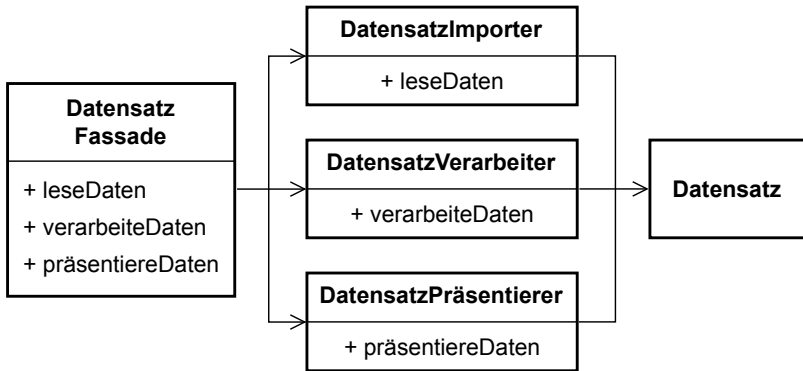


Betreuer



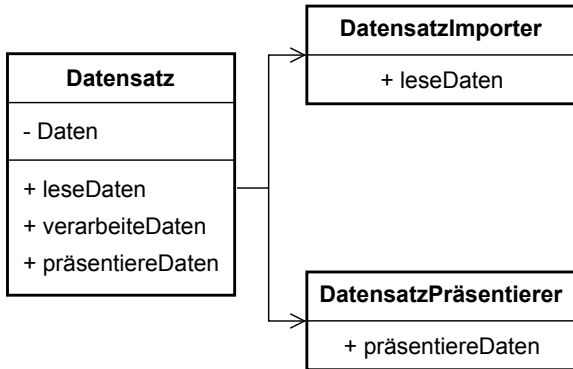
Beispiele für seinen Einsatz

Drei Lösungsmöglichkeiten (2): Fassade zur einfacheren Nutzung



Beispiele für seinen Einsatz

Drei Lösungsmöglichkeiten (3): Auslagerung von Teilaspekten



- ▶ vollständige Separation in Klassen
 - Verantwortlichkeiten vollständig voneinander getrennt
 - Die Klasse „Datensatz“ ist eine reine Datenstruktur.
 - kann mitunter unübersichtlich werden
 - ▶ Fassade zur einfacheren Nutzung
 - Lösung für das Problem zu vieler getrennter Klassen
 - erhält die Trennung nach Verantwortlichkeiten aufrecht
 - Zahl der Verantwortlichkeiten meist zeitlich konstant
 - ▶ Auslagerung von Teilaspekten
 - Daten und wichtigste Geschäftslogik bleiben zusammen
 - Fassade für ausgelagerte weitere Verantwortlichkeiten
- ☞ Umsetzung abhängig von der konkreten Situation

- ▶ eines der einfachsten Prinzipien...
 - konzeptionell einfach verständlich
- ▶ ...und eines der am schwierigsten richtig anzuwendenden
 - erfordert viel Erfahrung und Überblick
 - Zu viel ist genauso schlecht wie zu wenig.
- ▶ Manchmal lassen sich Kopplungen nicht ganz vermeiden.
 - wichtig: alle Abhängigkeiten weisen weg von dieser Klasse
- ☞ Verantwortlichkeiten korrekt voneinander zu trennen, ist ein zentraler Aspekt jeglicher Software-Architektur.
- ☞ Das Konzept wird uns auch bei den anderen Prinzipien immer wieder begegnen.

“ *Gather together those things that change
at the same times and for the same reasons.
Separate those things that change
at different times or for different reasons.*

– Robert C. Martin

- ▶ SOLID-Prinzipien ursprünglich für Klassen entwickelt
 - lassen sich auf Komponenten und die Architektur des Gesamtsystems genauso anwenden
- ▶ „Achse der Veränderung“ (*axis of change*)
 - auf Systemebene verantwortlich für (harte) Grenzen zwischen einzelnen Schichten der Architektur

“ *An axis of change is an axis of change only if the changes actually occur.*

– Robert C. Martin

- ▶ nur anwenden, wenn die Verantwortlichkeiten sich tatsächlich unabhängig voneinander ändern
 - hängt immer vom gegebenen Kontext ab
 - erfordert nötigen Weitblick, Kenntnis und Vertrautheit
- ▶ Prinzipien generell immer nur anwenden, wenn das zugehörige Symptom auftritt
 - gilt für alle Prinzipien
 - führt ansonsten oft zu unnötig komplexem Code



Zentrale Aspekte

- Ein Modul sollte nur Verantwortung gegenüber genau einem Akteur haben.
- Jede Verantwortlichkeit ist eine (potentielle) Quelle für Veränderungen.
- Verantwortlichkeiten richtig zu trennen, ist ein zentraler Aspekt jeglicher Software-Architektur.
- Trennung der Verantwortlichkeiten ist nur dann wichtig, wenn unabhängige Änderungen real auftreten.
- Eines der einfachsten Prinzipien für Software-Architektur – und eines der am schwersten korrekt umzusetzenden.