Programmierkonzepte in der Physikalischen Chemie

26. Interface-Segregation-Prinzip

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2016/17

Zentrale Aspekte

UNI FREIBURG

- Keine Klasse sollte Methoden implementieren müssen, die sie nicht nutzt.
- Werden mehrere Schnittstellen benötigt, sollten sie getrennt als abstrakte Klassen definiert werden.
- Symptome für den Einsatz:
  unnötige Kopplung und unnötige Komplexität
- sorgt für Entkopplung und übersichtlicheren Code und damit für Flexibiltät, Wiederverwendbarkeit, Wartbarkeit
- hängt eng zusammen mit dem Open-Closed-Prinzip und dem Dependency-Inversion-Prinzip

Übersicht

UNI FREIBURG

Das Interface-Segregation-Prinzip

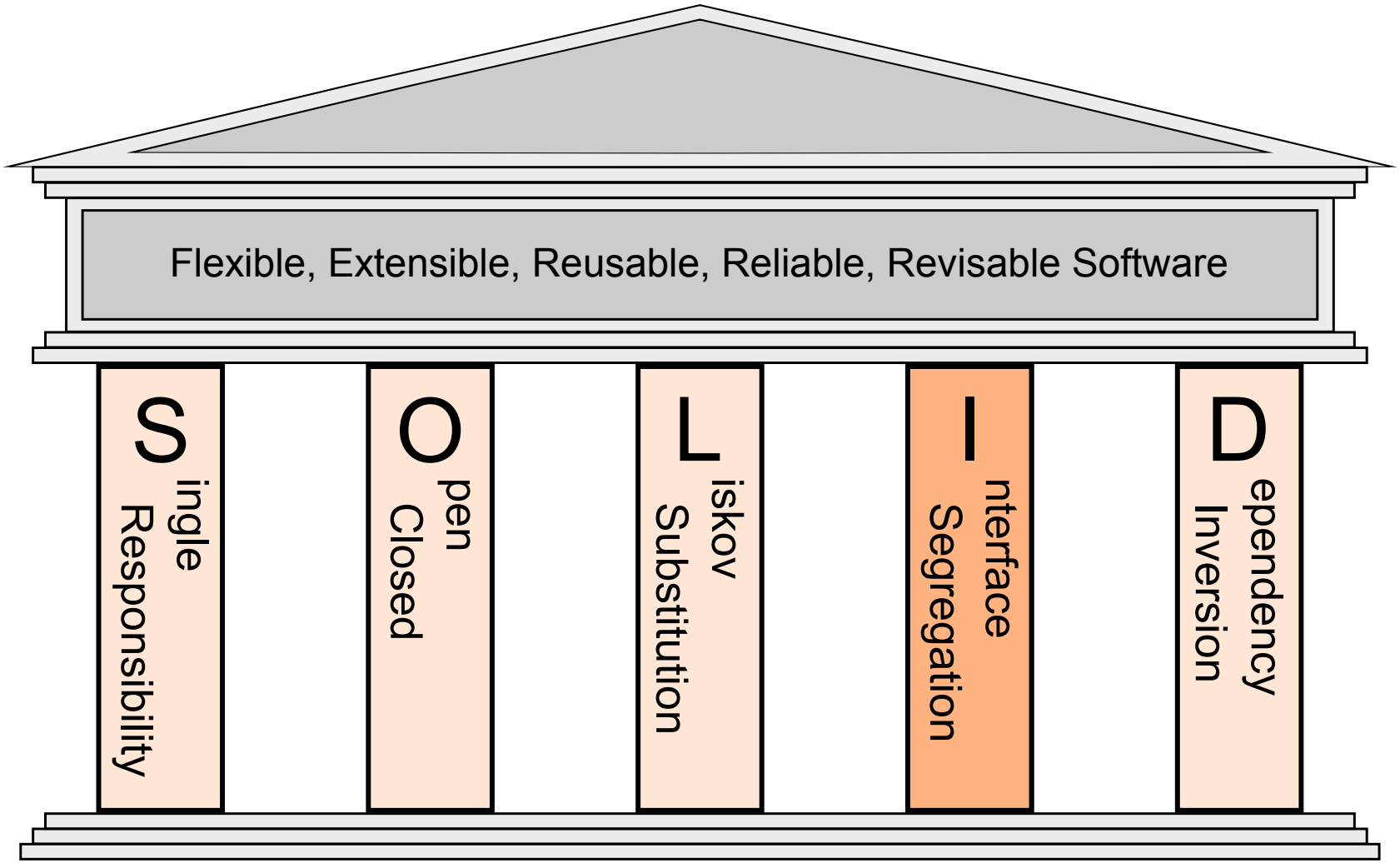
Symptome, die für seinen Einsatz sprechen

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

Das Interface-Segregation-Prinzip

Übersicht über die fünf Prinzipien



Das Interface-Segregation-Prinzip

Schnittstellen sollten nur aus notwendigen Methoden bestehen

UNI FREIBURG

> *Clients should not be forced to depend on methods that they do not use.*
>
> – Robert C. Martin

- Schnittstellen abstrakter Klassen sollten minimal sein.
  - Die Methoden der abstrakten Klasse sollten *alle* in *jeder* abgeleiteten Klasse verwendet werden.
- Objekte können mehr als eine Schnittstelle benötigen.
  - Jede Schnittstellte sollte als einzelne abstrakte Klasse mit zusammengehörenden Methoden definiert werden.
  - Nutzung über Adapter oder Mehrfachvererbung

Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 137

Symptome, die für seinen Einsatz sprechen

Unnötige Kopplung zwischen Einheiten, unnötige Komplexität

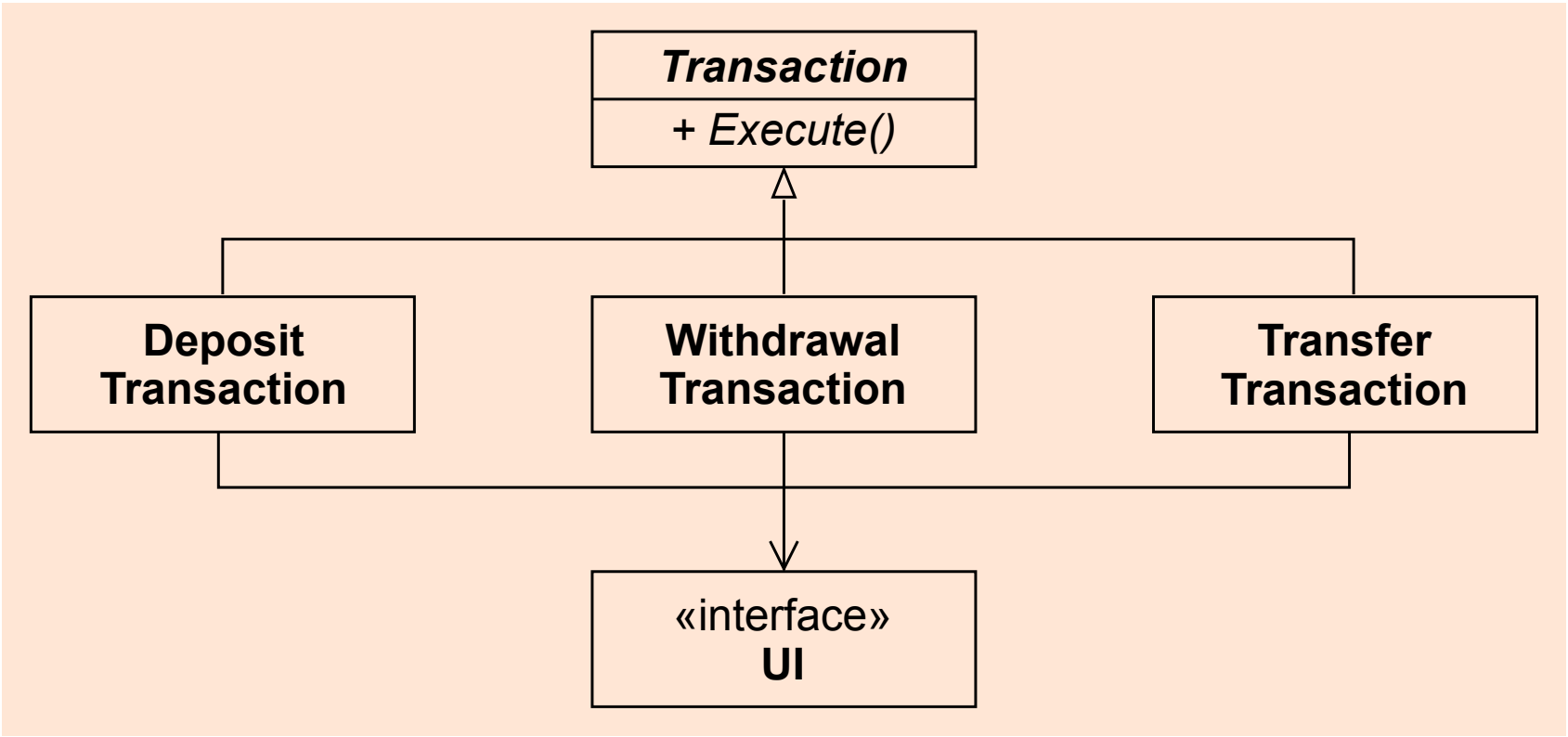
UNI FREIBURG

- unnötige Komplexität
  - Eine abgeleitete Klasse muss Methoden implementieren, die sie gar nicht braucht/brauchen kann.
  - kann das Liskov-Substitutionsprinzip verletzen
- unnötige Kopplung
  - Nutzer von Schnittstellen machen mitunter eine Änderung der Schnittstelle notwendig.
  - Änderungen wirken sich bei gekoppelten Schnittstellen auf viele unbeteiligte Objekte/Klassen aus.

☛ Trennung der Zuständigkeiten:
unterschiedliche Nutzer bekommen eigene Schnittstellen.

Beispiele für seinen Einsatz

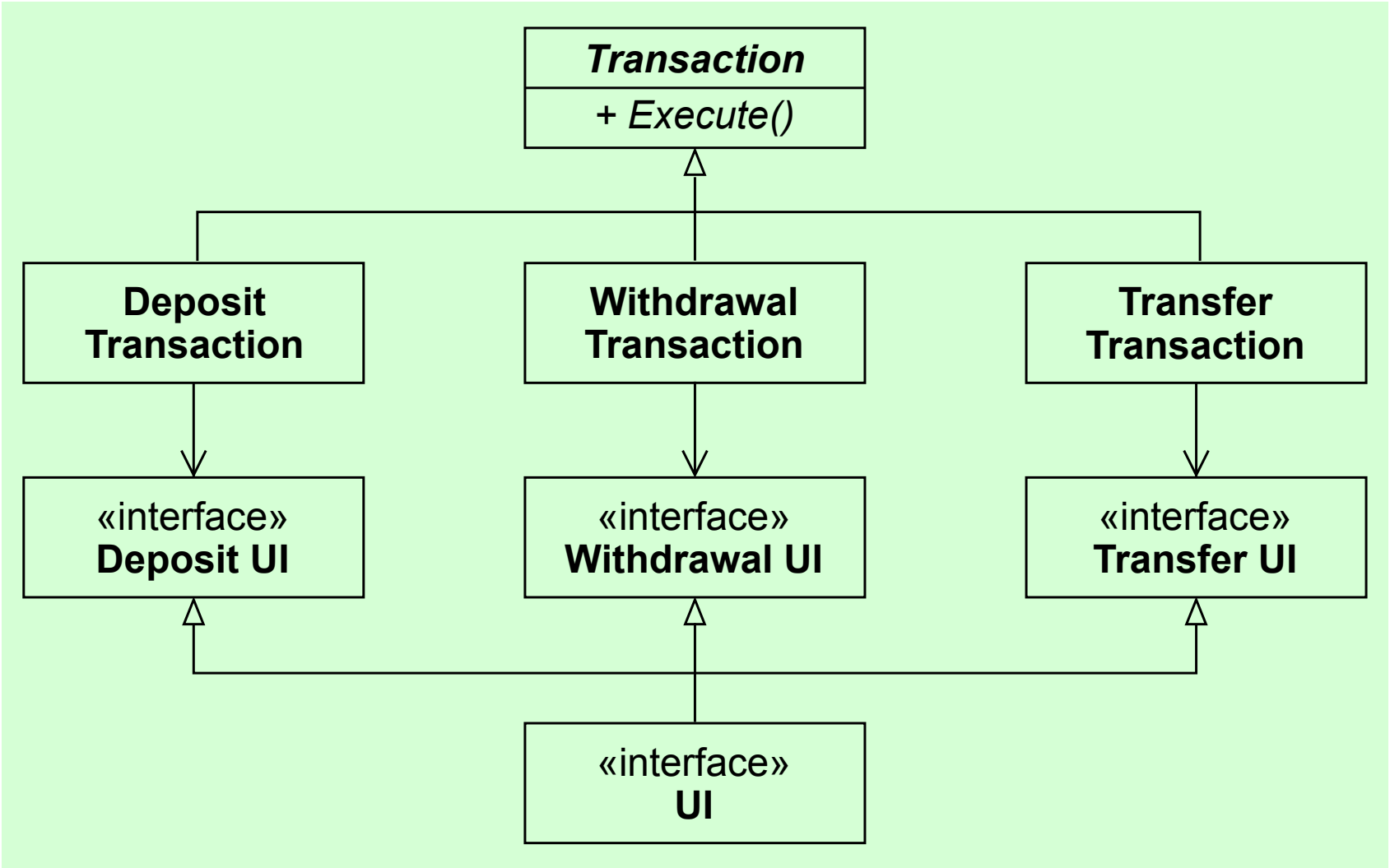
Das klassische Beispiel: der Geldautomat



- Jede Transaktion verwendet Methoden von `UI`, die keine andere Transaktion verwendet.

Beispiele für seinen Einsatz

Das klassische Beispiel: der Geldautomat



Beispiele für seinen Einsatz

Das klassische Beispiel: der Geldautomat

UNI FREIBURG

Situation vor Verwendung des Prinzips

- `UI` enthält viele unabhängige Methoden.
- Änderungen an `UI` beeinflussen jede Transaktion.

Situation bei Verwendung des Prinzips

- Entkopplung der Transaktionen von der Nutzerschnittstelle
  - hängen nur noch von ihren spezifischen Schnittstellen ab
  - Änderungen einer Transaktion oder neue Transaktionen haben keine Auswirkungen auf andere Transaktionen.
- Umkehr der Abhängigkeiten
  - Die Nutzerschnittstelle hängt von den Transaktionen ab, nicht die Transaktionen von der Nutzerschnittstelle.
  - Vorgriff auf das Dependency-Inversion-Prinzip

Bedeutung im Gesamtkontext

Hilfreich für die Modularität des Gesamtsystems

UNI FREIBURG

- Entkopplung
  - Einzelne Module lassen sich wiederverwenden.
  - Änderungen der Funktionalität bleiben lokal.
- übersichtlicherer Code
  - Klassen implementieren nur das Notwendige.
  - erhöht die Lesbarkeit und damit die Wartbarkeit
- Zusammenhang mit dem Single-Responsibility-Prinzip
  - Strategie zum Umgang mit Klassen, die mehr als eine Aufgabe erfüllen müssen
  - Jede Einzelaufgabe wird in eine Schnittstelle ausgelagert.

☛ sorgt für Flexibilität, Wiederverwendbarkeit, Wartbarkeit

Zentrale Aspekte

- Keine Klasse sollte Methoden implementieren müssen, die sie nicht nutzt.
- Werden mehrere Schnittstellen benötigt, sollten sie getrennt als abstrakte Klassen definiert werden.
- Symptome für den Einsatz:
  unnötige Kopplung und unnötige Komplexität
- sorgt für Entkopplung und übersichtlicheren Code und damit für Flexibiltät, Wiederverwendbarkeit, Wartbarkeit
- hängt eng zusammen mit dem Open-Closed-Prinzip und dem Dependency-Inversion-Prinzip