

Verteilte Systeme und Komponenten

S.O.L.I.D.-Prinzipien

Fünf zentrale Designprinzipien

Roland Gisler



Inhalt

S.O.L.I.D. fasst **fünf** wichtige Designprinzipien zusammen:

- **S**ingle **R**esponsibility **P**inciple (SRP)
- **O**pen **C**losed **P**inciple (OCP)
- **L**iskov **S**ubstitution **P**inciple (LSP)
- **I**nterface **S**egregation **P**inciple (ISP)
- **D**ependency **I**nversion **P**inciple (DIP)

Lernziele

- Sie kennen die fünf grundlegenden S.O.L.I.D.-Designprinzipien.
- Sie können die Prinzipien anhand von Beispielen erklären.
- Sie können die Prinzipien in eigenen Entwürfen anwenden.

Ziel der **S.O.L.I.D.-Prinzipien**

- Durch die Einhaltung der fünf fundamentalen **S.O.L.I.D.-Prinzipien** erreicht man ein qualitativ besseres und schöneres Design.
- Besseres Design heisst konkret:
 - Höhere Wiederverwendbarkeit
 - Leichtere Verständlichkeit / bessere Lesbarkeit
 - Verbesserte Testbarkeit
 - Vereinfachte Wartung
 - Verbesserte Erweiterbarkeit
 - Leichteres Refactoring

SRP

Single **R**esponsibility **P**rinciple

Single Responsibility Principle (SRP)

- Hauptziele
 - Eine Klasse* soll nur **eine** Verantwortlichkeit haben.
 - Eine Klasse soll nur **einen** Grund zur Änderung haben.
- Einhaltung von SRP hat eine hohe Kohäsion zur Folge.
 - Es kommt und bleibt zusammen, was zusammen gehört.
- Wird SRP verletzt, ergibt sich umgekehrt eine hohe Kopplung.
 - ➔ Höhere Komplexität, schlechtere Wart- und Erweiterbarkeit.
- SRP gilt aber auch auf den Ebenen der Komponenten, Schichten, Teilsysteme: Unterschiedliche Abstraktionsebenen!

* Beispiel für ein beliebiges Softwareartefakt, gilt sinngemäss auch für Modul, Package, Methode etc.

SRP ist eine fundamentale Grundlage von OOD

- Das Single Responsibility Prinzip gilt als eines der fundamentalen Prinzipien des objektorientierten Designs.
 - Als Konzept relativ einfach, wird schnell verstanden.
- Einhaltung von SRP liefert im Design typisch viel **mehr** und dafür aber **kleinere** Klassen
 - Das ist deutlich besser als wenige, grosse Klassen!

Aber: SRP ist eines der am häufigsten verletzten Prinzipien!

- Wir treffen sehr häufig auf Funktionen und Klassen, die (viel) zu viele Aufgaben auf einmal erfüllen wollen (bis hin zu «Gott»-Klassen).
- Wir nutzen Tools und Werkzeuge, die zu viel auf einmal machen (wollen) und eine sehr starke Abhängigkeit produzieren.

Beispiel: Modem

- Ein altes Beispiel, in Anlehnung an Tom DeMarco, einem erklärten SRP-Anhänger (und Vorreiter von SA/SD):



```
interface Modem {  
    void dial(String phoneNumber);  
    void hangup();  
    void send(char data);  
    char receive();  
}
```



- Eine schmale und schlanke Schnittstelle!
- Ein schönes Beispiel für «gute» Objektorientierung!
- Ähm, oder vielleicht doch nicht?

Modem – Beispiel

- Mögliche Gründe für eine Änderung sind:
 - Änderung des Wahlvorgangs:
z.B. die automatische Wahlwiederholung wenn besetzt.
 - Änderungen in der Datenübertragung:
z.B. die Vergrößerung des Datenbuffers.

➔ Das sind **zwei** Gründe!
- Das **Single Responsibility Prinzip** sagt aber:
Es soll nur **einen** Grund für Änderungen geben!
- Und tatsächlich: Der Verbindungsauf- und -abbau hat (funktional) absolut **nichts** mit der Datenübertragung zu tun!
- Prinzip der **Single Responsibility Principle** ist hier somit **verletzt**!

Modem – Lösung mit SRP

- Darum sollte man die Schnittstellen auftrennen:

```
interface Transmit {  
    void send(char character);  
    char receive();  
}
```



```
interface Connection {  
    void dial(String phoneNumber);  
    void hangup();  
}
```



- Schnittstellen werden schmaler, die Wiederverwendbarkeit höher.
- Wie ist das beim Logger-Projekt VSK: **Logger** und **LoggerSetup**

SRP - Zusammenfassung

- Unix-Philosophie:
«Tu nur ein Ding, genau ein Ding, das aber richtig!»
- Eine Klasse hat möglichst nur **eine** Zuständigkeit.
- Eine Klasse hat somit nur **einen** Grund zur Änderung.
- Änderungen oder Erweiterungen sollten sich auf möglichst wenig Klassen beschränken.
 - Die hohe Kohäsion bleibt erhalten.
- Viele kleine Klassen sind besser als wenige grosse Klassen.
- Wichtiger Nebeneffekt: Stark verbesserte Testbarkeit!

OCP

Open **C**losed **P**rinciple


Open Closed Principle

- Grundidee: Eine Klasse* soll «offen» für Erweiterungen, aber «geschlossen» gegenüber Modifikationen sein.
- **Offen** für Erweiterungen:
Design ist für eine einfache und sichere Erweiterbarkeit ausgelegt, beispielsweise durch Einsatz des Strategy-Pattern (GoF):
Erweiterung durch einfaches Anlegen einer **neuen** Klasse.
- **Geschlossen** für Änderungen:
Design ist so ausgelegt, dass **bestehende** Methoden und Klassen bei einer Erweiterung möglichst **nicht** verändert werden müssen.
- Motivation: **Reduktion des Risikos neue Fehler einzubauen.**

* Beispiel für ein beliebiges Softwareartefakt, gilt sinngemäss auch für Modul, Package, Methode etc.

Open Closed Principle – Beispiel

```
public double calc(Operation op, double arg1, double arg2) {  
    double result = 0.0;  
    switch (op) {  
        case Addition:  
            result = arg1 + arg2; break;  
        case Subtraktion:  
            result = arg1 - arg2; break;  
        default:  
            throw new IllegalArgumentException(); break;  
    }  
    return result;  
}
```



- Was passiert, wenn eine neue Operation ergänzt werden soll?
→ Es besteht ein hohes Risiko bei der Erweiterung einen Fehler einzubauen. Klassiker: Das **break** zu vergessen.

Bessere Lösung

- Auslagern der erweiterbaren Funktionen in → Strategien, anstatt die bestehende Funktion zu verändern.

```
interface Operation {  
    double calc(double arg1, double arg2);  
}
```



```
double calc(Operation op, double arg1, double arg2) {  
    return op.calc(arg1, arg2);  
}
```

- Erweiterung durch neue Strategien, welche das Interface implementieren.
 - Die Operation ist nun ein Interface.
 - Das ursprüngliche **switch**-Statement entfällt vollständig!

OCP - Zusammenfassung

- Eine Software-Entität soll offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein.
- Mit OCP **senkt** man das **Risiko** neue Fehler einzubauen, weil man bestehenden Code **nicht** (oder weniger) ändern muss.
- Häufig über Einsatz des Strategy-Patterns (GoF) erreicht.
 - Eliminiert oder reduziert die **if/switch**-Statements:
<http://www.industriallogic.com/xp/refactoring/conditionalWithStrategy.html>
- OCP ist ein sehr wirkungsvolles, aber anspruchsvolles Prinzip!

LSP

Liskov Substitution Principle

Liskov Substitution Principle (LSP)

- Liskov'sches Substitutionsprinzip:
«Eigenschaften, die anhand der Spezifikation des vermeintlichen Typs eines Objektes bewiesen werden können, sollen auch dann gelten, wenn das Objekt einer Spezialisierung dieses Typs angehört.»
- Etwas einfacher formuliert:
«Subtypen sollten sich so verhalten wie ihre Basistypen.»
- Noch etwas einfacher formuliert:
In spezialisierten Klassen nur Methoden ergänzen oder für Erweiterung überschreiben, aber **nie** fundamental verändern!

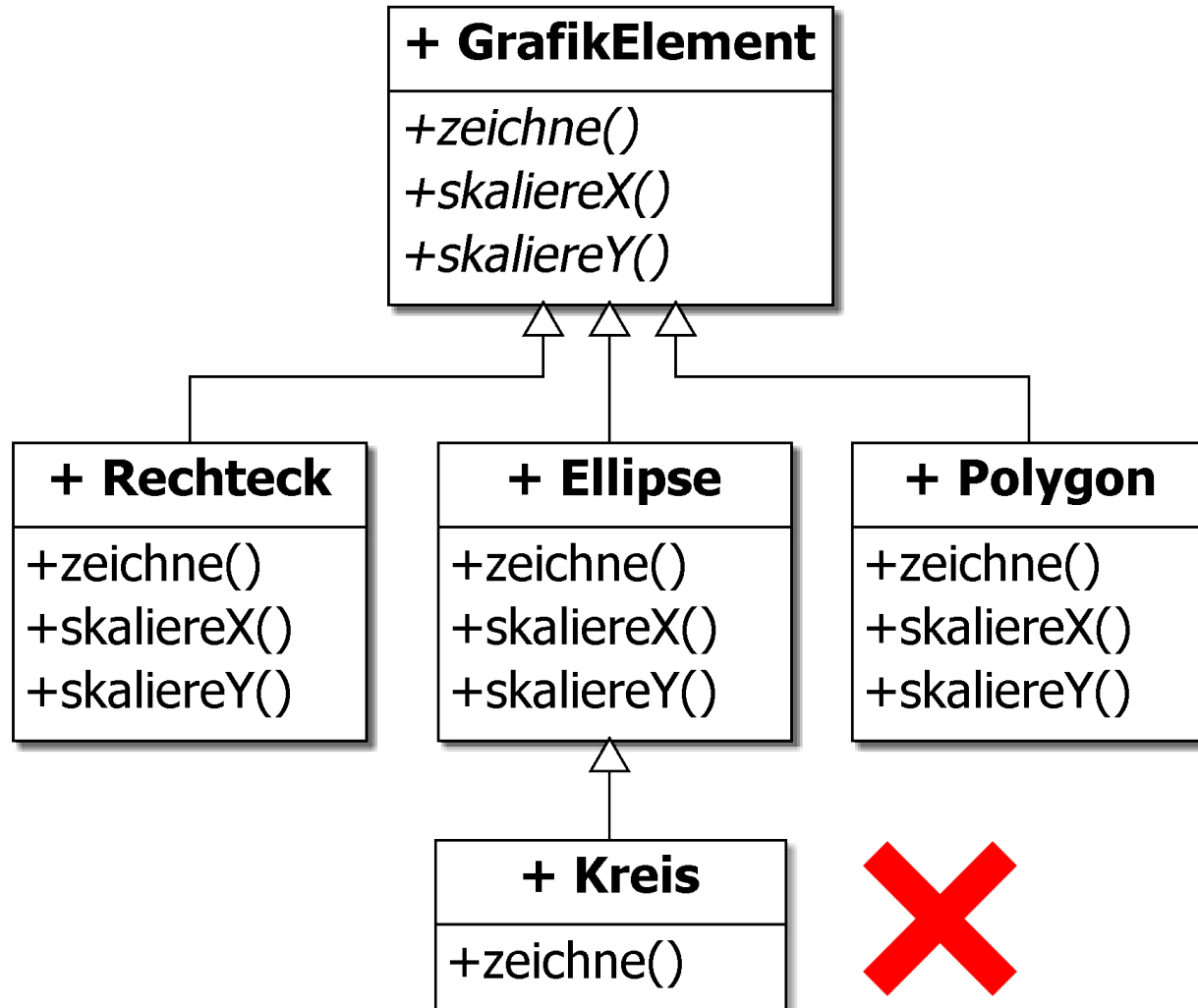
Barbara Jane Huberman Liskov

- Barbara Jane Huberman Liskov
- Professorin für Elektrotechnik und Informatik am MIT.
- 1968 erhielt sie an der Stanford University als erste Frau in den USA den Titel eines Ph.D. in Informatik.
- 2008 erhielt sie als erst zweite Frau (nach Frances E. Allen) den Turing Award.
- Gemeinsam mit Jeannette Wing entwickelte sie 1993 das für die OOP bedeutsame Liskov'sche Substitutionsprinzip.



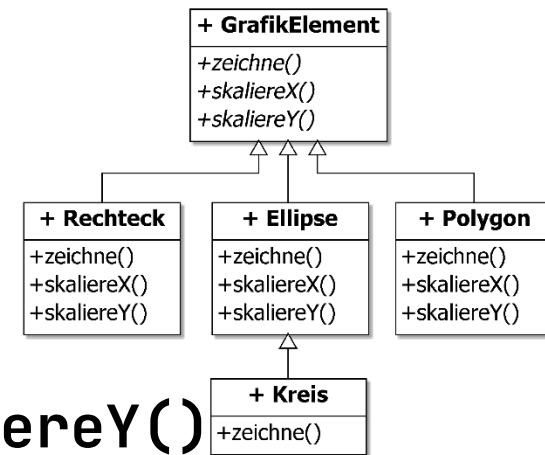
LSP-Klassiker: Kreis-Ellipse Problem

- Annahme: Die **skaliere*()**-Methoden werden später ergänzt.
 - Wo liegt das Problem?

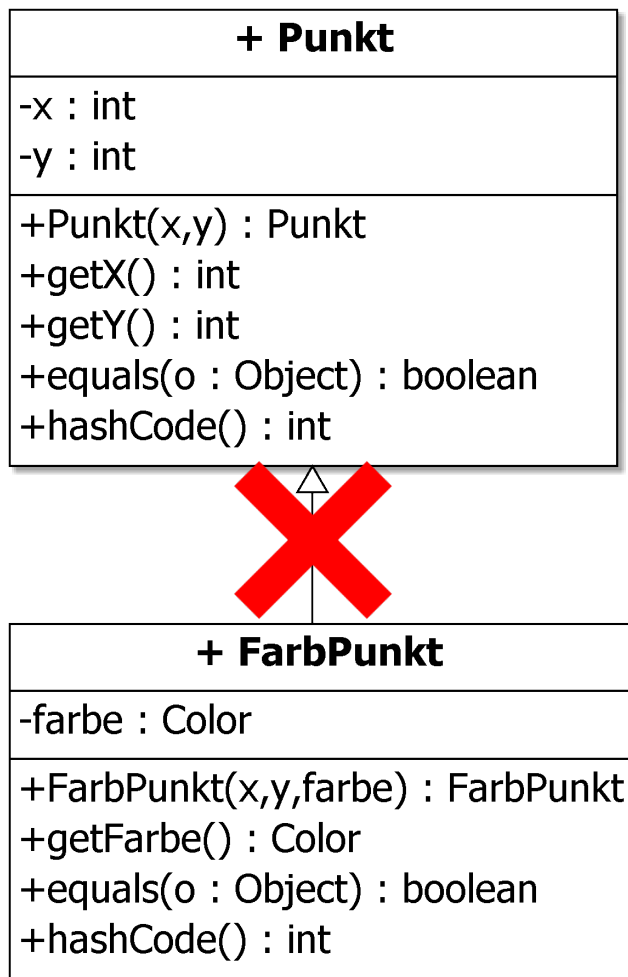


Kreis-Ellipse Problem

- Kreis scheint eine Spezialisierung der Ellipse mit der zusätzlichen Bedingung $r_x=r_y$ zu sein.
 - Die Ergänzung von **skaliereX()** und **skaliereY()** ist weder für Rechteck, Ellipse noch Polygon ein Problem.
➔ Aber was ist mit dem Kreis?
 - Der Kreis erbt alle Methoden von der Ellipse. Bei einem Kreis dürfen r_x und r_y aber nicht mehr unabhängig voneinander skaliert werden!
 - Die Anforderung: «Die Achsen können unabhängig voneinander skaliert werden.» stimmt für den Kreis nicht mehr!
- ➔ **Das Liskov'sche Substitutionsprinzip ist hier verletzt!**



Das hatten wir doch auch schon mal!



- Problem bei der Einhaltung des **equals**-Contracts bei der Spezialisierung nach **FarbPunkt**.
- Anforderung der **Symmetrie**: Wenn **punkt.equals(farbpunkt)** dann ist auch **farbpunkt.equals(punkt)**.
- Verhält sich ein **farbiger** Punkt genauso wie ein normaler Punkt? **Nein**, weil er berücksichtigt eventuell noch die Farbe in seinem Verhalten!
- **LSP ist hier meistens auch verletzt!**
 - Letztlich Abhängig vom Kontext!

LSP - Zusammenfassung

- Den Sinn von Vererbung / Spezialisierung immer kritisch verifizieren: Subtypen sollten sich so verhalten wie ihre Basistypen.
- Verifiziere Entscheide mit folgenden Sätzen:
 - Subtyp ist ein (**is-a**) Basistyp.
 - Subtyp verhält sich (**behaves-as**) wie ein Basistyp.
- **FCoI**: Meistens ist die Komposition der Vererbung vorzuziehen!
- Macht die Implementation von `equals()` Schwierigkeiten, sollte man unbedingt die Vererbung hinterfragen!
- Tipp: Vererbung konsequent verhindern (**final**)!
 - Nur wo sinnvoll und vorgesehen ein explizites Design für Spezialisierungen (Design for inheritance or else prohibit it).

ISP

Interface **S**egregation **P**rinciple

Interface Segregation Principle

- Clean Code: Interface Segregation Principle (ISP)
 - Artikel von Uncle Bob (Robert C. Martin, 1996):
<http://www.objectmentor.com/resources/articles/isp.pdf>
 - Segregation: «Entmischung» → Trennung
- Schnittstellen strikt von Details der Implementation trennen.
- Schnittstellen sollten sauber voneinander getrennt sein.
 - Keine Überschneidungen.
 - Keine Population von Schnittstellen.
 - Keine «fat»-Schnittstellen.
- Eine Schnittstelle soll eine hohe Kohäsion aufweisen.
 - Nur Methoden, die wirklich zusammen gehören.
- Nebenbei: Konzept des «design by interface» hilft hier.

Interface Segregation Principle

- Klienten sollten nur von Schnittstellen abhängig sein, die sie wirklich brauchen.
- Gibt es verschiedenartige Klienten eines Systems, sollte jeder Typ von Klient seine eigene Schnittstelle haben.
 - Abhängigkeiten minimieren → Kopplung minimiert.
 - Hat auch positiven Einfluss auf die Sicherheit.
- Basisklassen sollten nichts von ihren Spezialisierungen wissen.
 - Abstrakte Basisklassen als Schnittstellen.
- Schnittstellen feingranular entwerfen.
 - Viele kleine Schnittstellen sind besser als wenige grosse Schnittstellen.

Refactorings für ISP

Refactorings zur Erreichung der Interface Segregation:

- Superklasse aus bestehender Klasse extrahieren.
 - Superklasse ist dann häufig abstrakt (Schnittstelle).
 - Gefahr der (unerwünschten) Population der Schnittstelle!
- Interface aus bestehender Klasse extrahieren.
 - Geringere Kopplung, vgl. Komposition statt Vererbung (FCoI).
 - Unabhängigkeit von Implementation.
- Bestehende Interfaces auftrennen.
 - Schmalere Schnittstellen, geringere Kopplung.
 - vgl. Separation of Concerns (**SoC**)
 - vgl. **SRP** – Beispiel: Interface für ein Modem)

ISP - Zusammenfassung

- Schnittstelle strikt von Details der Implementation trennen.
 - Ist mit Java **einfach**: Weil wir haben Interfaces!
- Schnittstellen sollen eine hohe Kohäsion haben.
- Kopplung zwischen Komponenten soll minimal sein.
- Viele kleine (schmale) Schnittstellen sind besser als eine zu grosse (fette) Schnittstelle.
- Population von Schnittstellen vermeiden
 - ➔ möglichst **keine** Vererbung von Schnittstellen!

DIP

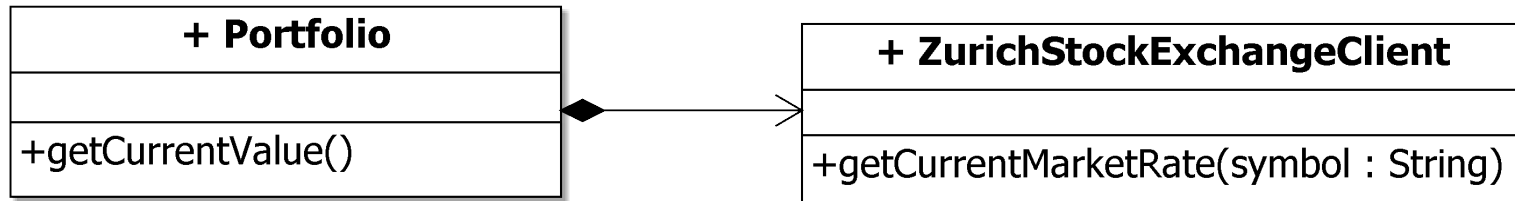
Dependency **I**nversion **P**rinciple

Dependency Inversion Principle (DIP)

- Ziel: Änderungen isolieren.
 - Artikel von Uncle Bob (Robert C. Martin, 1996):
<http://www.objectmentor.com/resources/articles/dip.pdf>
- High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern allenfalls beide von Interfaces.
 - High-Level: Hoher Abstraktionsgrad, Konzeptionell.
 - Low-Level: Konkrete, detailbehaftete Implementation.
 - Siehe auch **Single Level of Abstraction (SLA)**
- Analog: Interfaces sollen nie von Details der Implementation abhängig sein, sondern allenfalls Implementationen von Interfaces.

Änderungen isolieren

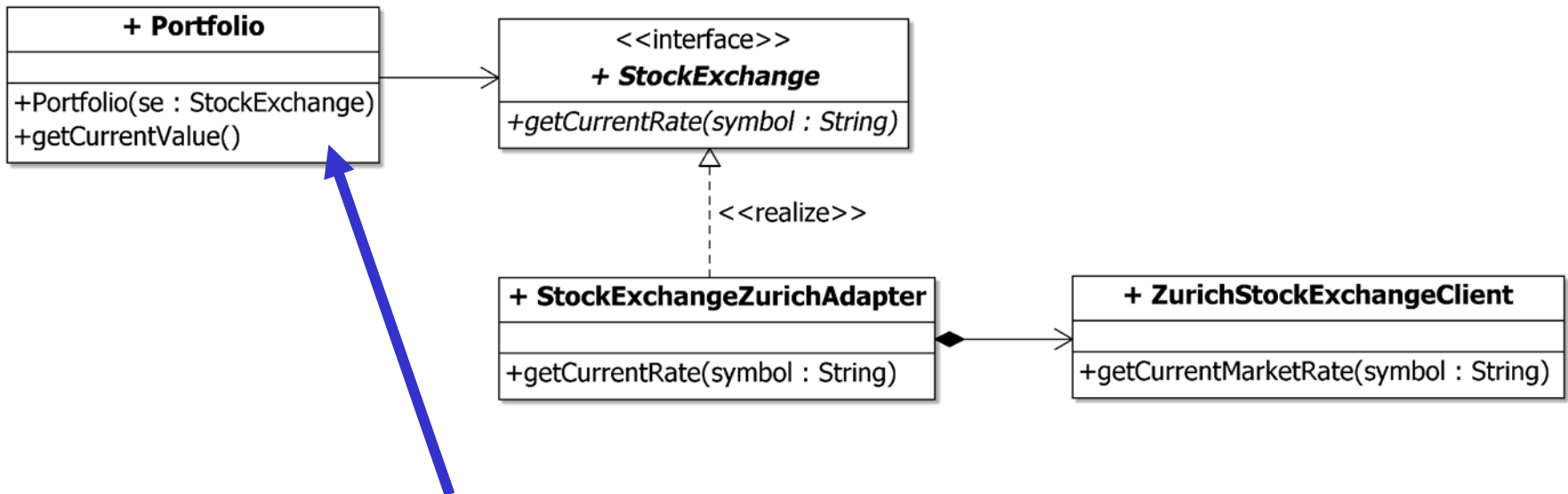
- Abstrakte Klassen und/oder Interfaces einführen, um die Auswirkungen von Implementationsdetails zu minimieren.
- Schlechtes Beispiel:
 - **Portfolio** und **ZurichStockExchangeClient**



- High-Level-Klasse **Portfolio** ist abhängig von Low-Level-Klasse **ZurichStockExchangeClient** → **Schlecht!**

Änderungen isolieren

- Bessere Lösung: **Portfolio** unabhängig von Implementationsdetails (**ZurichStockExchangeClient**)



- Nebenbei mit Dependency Injection (DI): Konstruktor mit **StockExchange** erlaubt einfachere Testbarkeit von **Portfolio**. (z.B. durch den Einsatz von ➔ Test Doubles).
- Nebenbei: Das entspricht dem **Adapter**-Pattern nach **GoF**!

DIP - Zusammenfassung

- High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern beide von Interfaces.
- Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces.
- Isolation von Klassen vereinfacht/ermöglicht die Testbarkeit, ggf. auch mit Einsatz von Test Doubles.
- Auflösung von Dependencies über **D**ependency **I**njection (**DI**).

S.O.L.I.D.

Schreiben Sie **SOLIDen** Code!

Vermeintlicher Angriff auf SOLID: C.U.P.I.D. - Prinzipien

- Im Jahr 2021 wurde von Daniel Terhorst-North eine 2017 entstandene «Provokation» unter dem Titel «Why every single element von SOLID is wrong» veröffentlicht.
- **C.U.P.I.D.** ist eine gute Ergänzung zu S.O.L.I.D. – aber kein Ersatz:
EP_52_CupidPrinzipien
(optionale Ergänzung)



Zusammenfassung – S.O.L.I.D.

- **Single Responsibility Principle (SRP)**
 - Spezialisierung von Separation of Concerns (SOC).
 - Ein Element soll nur einen Grund für Änderungen haben.
- **Open Closed Principle (OCP)**
 - Ein Element soll offen für Erweiterungen sein, aber geschlossen gegen Modifikationen.
 - Neue Funktionalitäten können ergänzt werden, ohne dass bestehender Code geändert werden muss.
- **Liskov Substitution Principle (LSP)**
 - Eine Spezialisierung verhält sich immer wie sein Basistyp.
 - Kann somit jederzeit den Platz des Basistyps einnehmen.

Zusammenfassung – S.O.L.I.D.

- **I**nterface **S**egregation **P**rinciple (ISP)
 - Clients sollen nicht mit Details belastet werden, die sie nicht benötigen.
 - Bewirkt eine lose Kopplung und eine hohe Kohäsion.
- **D**ependency **I**nversion **P**rinciple (DIP)
 - Highlevel Klassen sollen nicht von Lowlevel Klassen abhängig sein, sondern beide von Interfaces.
 - Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces.

Fragen?