Physikalische Chemie, Universität Rostock

**Vorlesung: Wissenschaftliche Softwareentwicklung**

**Wintersemester 2024/25**

Dr. habil. Till Biskup

— Glossar zu Lektion 25: „Dependency-Inversion-Prinzip“ —

---

*Hinweis: Die nachfolgend genannten Begriffe und Definitionen erheben keinen Anspruch auf formale Korrektheit, sondern dienen lediglich dem besseren Verständnis der in der Vorlesung behandelten Themen und sind im jeweiligen Kontext zu sehen. Mehrfache, voneinander abweichende Definitionen in unterschiedlichen Kontexten sind daher möglich. Englische Begriffe werden zwar nach Möglichkeit übersetzt, erscheinen aber ggf. unter ihrem englischen Namen in der Liste. Verweise untereinander sind durch ↑ gekennzeichnet.*

**Abhängigkeit** *dependency*, im Quellcode durch explizite Nennung hervorgerufene ↑Kopplung von Programmteilen (↑Funktionen, ↑Objekte, …), die dazu führt, dass der aufgerufene Programmteil nicht mehr ohne Veränderung des aufrufenden Teils verändert werden kann.

**abstrakte Fabrik** *abstract factory*, ↑abstrakte Schnittstelle zur Erzeugung einer Familie von ↑Objekten. Die konkreten ↑Klassen der zu erzeugenden Objekte werden nicht näher festgelegt und sind dem Nutzer unbekannt. Eines der „klassischen“ ↑Entwurfsmuster. Eine Anwendung wird zumindest immer ein Objekt einer konkreten Implementation (Klasse) aufweisen, die von der abstrakten Fabrik erbt (↑Vererbung).

**abstrakte Klasse** ↑Klasse, die zunächst einmal nur eine ↑Schnittstelle liefert und nur (abstrakte) ↑Methoden ohne Implementierung enthält.

**abstrakte Schnittstelle** *abstract interface*, ↑abstrakte Klasse, die in Sprachen, die keine ↑Mehrfachvererbung unterstützen, explizit als ↑Schnittstelle (*interface*) definiert wird, so dass eine Klasse von mehreren abstrakten Schnittstellen „erben“ kann.

**Abstraktion** Nach Edsger Dijkstra [1] das einzige mentale Werkzeug, das es erlaubt, eine große Vielzahl von Fällen abzudecken. Zweck der Abstraktion ist es nicht, vage zu sein, sondern im Gegenteil ein neues Bedeutungsniveau zu schaffen, das präzise Beschreibungen erlaubt.

**Attribut** im Kontext der ↑objektorientierten Programmierung eine Variable, die innerhalb einer ↑Klasse definiert wird. ↑Methoden operieren auf den Attributen einer ↑Klasse bzw. dem daraus erzeugten ↑Objekt.

**Dependency Inversion** Umkehr der ↑Abhängigkeiten gegenüber der intuitiven Implementierung. Abhängigkeiten sollten häufig entgegen dem ↑Kontrollfluss verlaufen.

**Dependency-Inversion-Prinzip** (DIP) Anwendung der ↑Dependency Inversion: Abstraktionen sollten nicht von Details abhängen. Umgekehrt sollten Details auf Abstraktionen aufbauen.

**DIP** ↑Dependency-Inversion-Prinzip, vgl. ↑SOLID

**Entwurfsmuster** *design patterns*, erprobte und bewährte Lösungen für wiederkehrende Probleme in der Softwareentwicklung. Beschreibungen miteinander kommunizierender ↑Objekte und ↑Klassen, die maßgeschneidert sind, um ein generelles Entwurfsproblem in einem bestimmten Kontext zu lösen. [2, S. 3] Entwurfsmuster liefern eine (↑abstrakte) Beschreibung des dahinterstehenden Konzepts und haben meist einen etablierten Namen, der

die Kommunikation erleichtert. Es gibt ganze Kataloge solcher Muster, und viele der ursprünglich beschriebenen Entwurfsmuster sind heute in vielen Programmiersprachen fest etabliert.

**Framework** Rahmenstruktur, Ordnungsrahmen; in der Softwaretechnik ein Programmiergerüst, das den Rahmen zur Verfügung stellt, innerhalb dessen der Programmierer eine Anwendung erstellt. Die Struktur der individuellen Anwendung wird u.a. durch die im Framework verwendeten ↑Entwurfsmuster beeinflusst. Das ↑Dependency-Inversion-Prinzip ist eine wesentliche Voraussetzung für den Entwurf flexibler, wiederverwendbarer Frameworks.

**Funktion** im Kontext der strukturierten Programmierung eine Liste von Anweisungen, die eine bestimmte Aufgabe erfüllt und der Programmiersprache unter einem festen Namen bekannt ist. Vgl. ↑Methode.

**Interface Segregation** Aufteilung der ↑Schnittstelle einer ↑Klasse oder eines ↑Moduls mit dem Ziel möglichst geringer ↑Kopplung und hoher ↑Kohäsion.

**Interface-Segregation-Prinzip** Anwendung der ↑Interface Segregation: Nutzer sollten nicht dazu gezwungen werden, von Methoden abzuhängen, die sie nicht verwenden.

**ISP** ↑Interface-Segregation-Prinzip, vgl. ↑SOLID

**Kapselung** *encapsulation*, ein ↑Objekt enthält Daten (↑Attribute) *und* zugehöriges Verhalten (↑Methoden) und kann beides nach Belieben vor anderen Objekten verstecken.

**Kern der Anwendung** Implementation des zugrundeliegenden ↑Modells in Software, d.h. Abbildung auf Code, zumeist in Form abstrakter ↑Klassen

**Klasse** *class*, im Kontext der ↑objektorientierten Programmierung die Blaupause für die Erzeugung eines ↑Objektes; Definition der Daten (↑Attribute) und des zugehörigen Verhaltens (↑Methoden).

**Kohäsion** *cohesion*, innerer Zusammenhalt; hier: Zusammenhang einzelner Aspekte einer Softwareeinheit zueinander. Ein Ziel der Softwareentwicklung ist starke Kohäsion (*strong/high cohesion*). Jede Einheit (z.B. ↑Methode, ↑Funktion, ↑Klasse) hat eine Aufgabe, und alle Teile dieser Einheit dienen dem Zweck, diese eine Aufgabe zu erfüllen.

**Kontrollfluss** *flow of control*, Reihenfolge des Aufrufs von Programmteilen (↑Funktionen, ↑Objekte, ...), um eine gegebene Aufgabe zu erfüllen.

**Kopplung** *coupling*, in Software der Grad der Verbindung zweier Komponenten; enge Bindung mehrerer Einheiten einer Software aneinander, so dass sie nicht unabhängig wiederverwendbar (bzw. ggf. auch nicht testbar) sind. Programmierkonzepte zielen generell auf eine lose Kopplung (*loose/low coupling*) einzelner Komponenten ab, da so die Wiederverwendbarkeit erleichtert wird.

**Liskov-Substitution** Einsatz von Subtypen anstelle ihrer Basistypen ohne Beeinträchtigung der Funktionalität.

**Liskov-Substitutionsprinzip** Anwendung der ↑Liskov-Substitution: Subtypen müssen durch ihre Basistypen ersetzbar sein. Grundlegendes Prinzip für die ↑Vererbung in der ↑objektorientierten Programmierung, das auf Barbara Liskov [3] zurückgeht.

**Lösungsraum** *solution domain*, Kontext der Programmierer eines Programms, Gegensatz zum ↑Problemraum. Namen aus dem Lösungsraum bestehen i.d.R. aus Begriffen aus der Welt der Programmierung.

**LSP** ↑Liskov-Substitutionsprinzip, vgl. ↑SOLID

**Mehrfachvererbung** (*multiple inheritance*) Eine ↑Klasse erbt (↑Vererbung) von mehr als einer ↑Superklasse. Wird von den wenigsten Programmiersprachen unterstützt, oftmals behilft man sich hier aber des Konzeptes einer ↑Schnittstelle (*interface*) (3.) und kann dann mehr als ein solches implementieren (bzw. davon erben). Konzeptionell lassen sich diese beiden Ansätze quasi identisch einsetzen.

**Methode** im Kontext der ↑objektorientierten Programmierung eine ↑Funktion, die innerhalb einer ↑Klasse definiert wird und auf den ↑Attributen einer ↑Klasse bzw. dem daraus erzeugten ↑Objekt operiert.

**Modell** sprachliche Formulierung des ↑Problemraums und seiner entscheidenden Zusammenhänge und Abläufe auf hohem Abstraktionsniveau; Summe der „Geschäftsregeln“

**Modul** Software-Einheit oberhalb von ↑Klassen, ↑Objekten und ↑Funktionen, aber unterhalb der Gesamtarchitektur (↑Softwarearchitektur) eines Systems. Module sind idealerweise so unabhängig voneinander wie möglich (↑Modularisierung). Entscheidend dafür sind lose ↑Kopplung und starke ↑Kohäsion.

**Modularisierung** Aufteilung der Gesamtaufgabe in kleinere Abschnitte. Die Aufteilung wird so lange fortgesetzt, bis die Lösung für den aktuellen Abschnitt unmittelbar in Form von Quellcode offensichtlich ist. Setzt die Definition von ↑Schnittstellen voraus.

**Objekt** *object*, im Kontext der ↑objektorientierten Programmierung der grundlegende Baustein eines Programms, bestehend aus den Daten (↑Attribute) und dem zugehörigen Verhalten (↑Methoden). Ein Objekt ist in diesem Kontext immer die Instanz einer ↑Klasse.

**objektorientierte Programmierung** (OOP) ein Programmierparadigma, bei dem Daten (Variablen zugewiesene Werte, als ↑Attribute bezeichnet) und Funktionen (↑Methoden), die auf diesen Daten (Attributen) operieren, eine Einheit bilden. Die in den ↑Attributen gespeicherten Daten lassen sich i.d.R. nur vermittelt durch (öffentlich zugängliche) ↑Methoden der ↑Klasse bzw. des daraus erzeugten ↑Objektes ansprechen. Es gibt eine klare Trennung zwischen öffentlicher ↑Schnittstelle und internen Verarbeitungsroutinen. Wichtige Vertreter objektorientierter Programmiersprachen sind Smalltalk, C++ und Java, aber auch Python.

**OCP** ↑Open-Closed-Prinzip, vgl. ↑SOLID

**Open Closed** Offenheit einer Software-Einheit für Erweiterungen bei gleichzeitiger Abgeschlossenheit gegenüber Abänderung

**Open-Closed-Prinzip** Anwendung von ↑Open Closed: Software-Einheiten (↑Klassen, ↑Module, ↑Funktionen etc.) sollten offen für Erweiterung, aber verschlossen gegenüber Abänderung sein.

**periphere Schnittstelle** ↑Schnittstelle zur ↑Peripherie eines Programms, sinnvollerweise in Form ↑abstrakter Schnittstellen realisiert. Bezogen auf Hardware Schnittstelle zu optionalen Geräten, die nicht integraler Bestandteil des Rechners sind.

**Peripherie** im Kontext der ↑Softwarearchitektur Randbereiche einer Anwendung wie Nutzerschnittstellen oder die ↑Persistenzschicht, die die Verbindung zur Außenwelt jenseits der Anwendung darstellen. Der ↑Kern der Anwendung sollte nie von der Peripherie abhängen.

**Persistenz** Fähigkeit, Daten (oder ↑Objekte) oder logische Verbindungen über lange Zeit (insbesondere über einen Programmabbruch hinaus) bereitzuhalten; benötigt ein nichtflüchtiges Speichermedium.

**Persistenzschicht** ↑Schicht in der ↑Softwarearchitektur eines Gesamtsystems, deren Aufgabe die ↑Persistenz ist.

**Polymorphie** *polymorphism*, „Vielgestaltigkeit“, ähnliche ↑Objekte können auf die gleiche Botschaft (den Aufruf einer gleichnamigen ↑Methode) in unterschiedlicher Weise reagieren.

**Problemraum** *problem domain*, Kontext der Fragestellung, die mit einem Programm (d.h. Software) angegangen werden soll, Gegensatz zum ↑Lösungsraum. Namen aus dem Problemraum verweisen i.d.R. auf Konzepte, mit denen die Anwender eines Programms vertraut sind (aber nicht notwendigerweise die Programmierer/Entwickler).

**SAP** ↑Stable-Abstractions-Prinzip

**Schichten** *layer*, abstrakteste (↑Abstraktion) Organisationsebenen eines Gesamtsystems im Kontext der ↑Softwarearchitektur.

**Schnittstelle** *interface*, Begriff mit mehreren leicht unterschiedlichen Bedeutungen; (1.) ↑Signatur einer ↑Methode. (2.) Im weiteren Sinne die Gesamtheit der öffentlichen ↑Attribute und ↑Methoden einer ↑Klasse bzw. eines ↑Objekts. Der Nutzer kennt nur die Schnittstelle, die Implementierung ist irrelevant und kann sich problemlos jederzeit ändern, solange die Funktionalität erhalten bleibt. Das dient der Trennung von Verantwortlichkeiten und ermöglicht ↑Modularisierung und ist in der Folge ein wesentlicher Aspekt der ↑Softwarearchitektur. (3.) In einer weiteren Bedeutung wird der Begriff (auch im Deutschen dann häufig mit seinem englischen Pendant) für (abstrakte) Klassen verwendet, die lediglich eine Schnittstelle (im Sinne von 2.) definieren. Das ist hauptsächlich dann von Bedeutung, wenn die Programmiersprache keine ↑Mehrfachvererbung unterstützt, aber das Implementieren von „*Interfaces*“. Vgl. ↑abstrakte Schnittstelle.

**SDP** ↑Stable-Dependencies-Prinzip

**Single Responsibility** Verantwortung gegenüber genau einer Sache und damit nur ein Grund für Änderungen

**Single-Responsibility-Prinzip** Anwendung der ↑Single Responsibility: eine ↑Klasse sollte nur einen Grund haben, sich zu ändern.

**Softwarearchitektur** Aufteilung eines größeren Projektes in einzelne kleinere Projekte bzw. Aufgaben (↑Modularisierung), Definition klarer ↑Schnittstellen und Anforderungen sowie der Interaktion der einzelnen Teile miteinander. Nach Robert C. Martin die Gestalt eines Systems, die ihm von seinen Entwicklern gegeben wird: Unterteilung des Systems in Komponenten, ihre Anordnung, und die Art ihrer Interaktion miteinander.

**SOLID** von Robert C. Martin eingeführtes Akronym aus den fünf Anfangsbuchstaben wichtiger Prinzipien für ↑Softwarearchitektur: ↑Single-Responsibility-Prinzip, ↑Open-Closed-Prinzip, ↑Liskov-Substitutionsprinzip, ↑Interface-Segregation-Prinzip, ↑Dependency-Inversion-Prinzip. Die von der „Gang of Four“ vorgestellten ↑Entwurfsmuster beruhen zu einem großen Teil (implizit) auf einer Umsetzung dieser fünf Prinzipien.

**SRP** ↑Single-Responsibility-Prinzip, vgl. ↑SOLID

**stabiler Hintergrund** *stable background*, konkrete Implementierungen in den Standardbibliotheken einer Programmiersprache, die gleichzeitig konkret sind und viele Abhängigkeiten aufweisen (weil alle Programme auf ihnen aufbauen), die sich aber höchst selten ändern. Ein Beispiel wäre eine Klasse `String` zur Repräsentation von Zeichenketten. Von solchen Komponenten abzuhängen ist gleichzeitig notwendig und ungefährlich. Deshalb stellen diese Abhängigkeiten auch keinen Verstoß gegen das ↑Stable-Abstractions-Prinzip dar.

**Stable-Abstractions-Prinzip** (SAP) Verallgemeinerung des ↑Open-Closed-Prinzips auf Ebene der Gesamtarchitektur: Eine Komponente sollte so abstrakt (vgl. ↑abstrakte Klasse) sein, wie sie stabil ist. ↑Abstraktion ist ein Schlüssel dazu, dass eine Softwarekomponente erweiterbar ist, ohne sie selbst verändern zu müssen.

**Stable-Dependencies-Prinzip** (SDP) Veränderbarkeit ist die Daseinsberechtigung für Software. ↑Abhängigkeiten sollten aber immer in Richtung der Stabilität zeigen: Niemand sollte von instabilen bzw. unbeständigen Elementen abhängen. Gemeinsam mit dem ↑Stable-Abstractions-Prinzip bildet das SDP die Verallgemeinerung des ↑Dependency-Inversion-Prinzips auf der Ebene der Komponenten: Abhängigkeiten sollten immer in Richtung zunehmender ↑Abstraktion zeigen.

**Subklasse** ↑Klasse, die von einer anderen Klasse (der ↑Superklasse) ↑Attribute und ↑Methoden erbt. Die Vererbung geht dabei i.d.R. über die nach außen hin sichtbare ↑Schnittstelle der Superklasse hinaus. Die Subklasse erbt von der ↑Superklasse häufig nur den „kleins-

ten gemeinsamen Nenner“ und implementiert die spezifische Funktionalität.

**Superklasse** ↑Klasse, von der andere Klassen (↑Subklassen) ↑Attribute und ↑Methoden erben. Die Vererbung geht dabei i.d.R. über die nach außen hin sichtbare ↑Schnittstelle der Superklasse hinaus. Superklassen implementieren bzw. definieren normalerweise nur das Notwendigste, sozusagen den „kleinsten gemeinsamen Nenner“. Alle spezifische Funktionalität wird in der ↑Subklasse implementiert.

**Typisierung** *typing*, Zuweisung eines Typs zu einem Objekt (im abstrakten Sinne) einer Programmiersprache, z.B. Ganzzahl (*integer*) oder Zeichenkette (*string*) im Fall einer Variable. ↑Abstraktion, die die Ausdrucksstärke von Programmiersprachen und Programmen deutlich erhöht, und die Überprüfung der Korrektheit erleichtert sowie Optimierungen ermöglicht. Typisierung kann explizit und implizit erfolgen. Darüber hinaus wird zwischen starker und schwacher Typisierung sowie zwischen statischer und dynamischer Typisierung unterschieden. Jede Art der Typisierung hat ihre Vor- und Nachteile, und unterschiedliche Programmiersprachen verwenden unterschiedliche Arten der Typisierung.

**Vererbung** *inheritance*, Weitergabe aller Eigenschaften (↑Attribute, ↑Methoden) von einer ↑Superklasse an eine ↑Subklasse. Die Subklasse ist vom gleichen Typ (↑Typisierung) wie die Superklasse, was wiederum die Grundlage der ↑Polymorphie ist. Änderungen der Superklasse wirken sich allerdings auf die Subklasse aus, die ↑Kapselung wird entsprechend geschwächt.

## Literatur


[1] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM* 15 (1972), S. 859–865.

[2] Erich Gamma u. a. *Design Patterns. Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.

[3] Barbara Liskov. Data Abstraction and Hierarchy. *ACM Sigplan Notices* 23.5 (1987), S. 17–34. DOI: 10.1145/62139.62141.