Physikalische Chemie, Universität Rostock

Vorlesung: Wissenschaftliche Softwareentwicklung

Wintersemester 2024/25

Dr. habil. Till Biskup

— Glossar zu Lektion 20: „Softwarearchitektur“ —

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.

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.

Abstraktionsebene Summe aller ↑Abstraktionen eines bestimmten Abstraktionsgrades.

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]

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.

Gesetz von Conway Organisationen, die Systeme entwerfen, werden immer Entwürfe erstellen, die sich an den Kommunikationsstrukturen dieser Organisationen orientieren. [3]

Gesetz von Demeter *Law of Demeter* (LoD), sprachunabhängige Regel für die objektorientierte Programmierung, die ↑Kapselung und ↑Modularisierung fördert. Objekte sollten nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren. Umgangssprachlich formuliert: „Sprich nur zu deinen nächsten Freunden“. Führt zum Begriff des „schüchternen Codes“. Lässt sich formalisieren und ist daher als Metrik geeignet.

Idiom unterschiedliche Bedeutungen; Linguistik: Spracheigentümlichkeit; Softwaretechnik: Umsetzung (Implementierung) abstrakter ↑Muster bzw. Lösung einfacher Aufgaben (auf niedrigster ↑Abstraktionsebene) in einer konkreten Programmiersprache

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 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.

Kristallkugel Nur in der Theorie funktionierendes Hilfsmittel für den Blick in die Zukunft, das u.a. hilfreich wäre, um Software bereits in ihrer Entstehung auf künftige Anforderungen hin auszulegen. Aufgrund anderer damit einhergehender Probleme ist die reale Funktionalität einer Kristallkugel nicht wünschenswert.

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 [4] 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

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.

Muster *pattern*, nach Christopher Alexander abstrakte Beschreibung eines wiederkehrenden Problems sowie einer generellen Lösung für dieses Problem, deren konkrete Ausgestaltung meist hochgradig individuell ist. In der Softwareentwicklung tauchen Muster auf unterschiedlichen Ebenen auf, angefangen bei ↑Idiomen über ↑Entwurfsmuster bis zu Mustern auf der Ebene der ↑Softwarearchitektur.

Nebenwirkung *side effect*, Wirkung; in der theoretischen Informatik die Veränderung des Programmzustands einer abstrakten Maschine. In der Praxis der Programmierung meist die Auswirkung einer Zuweisung eines Wertes zu einer Variablen, die außerhalb des konkret betrachteten Kontextes liegt.

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.

orthogonales Design größtmögliche Unabhängigkeit einzelner Komponenten einer Software voneinander. Der Begriff kommt ursprünglich aus der Mathematik und lässt sich elegant an zwei orthogonalen Vektoren verdeutlichen: Bewegung parallel zu einem Vektor verändert nicht die Projektion entlang des anderen Vektors. Orthogonales Design in Software führt zu kompaktem Design selbst komplexer Zusammenhänge. Wichtige Aspekte sind starke ↑Kohäsion und geringe ↑Kopplung. Rein orthogonales Design ist frei von ↑Nebenwirkungen: Jede Operation verändert nur genau eine Sache ohne Einfluss auf andere Aspekte. Darüber hinaus gibt es jeweils nur genau einen Weg, eine Eigenschaft eines Systems zu verändern.

Paradigma nach Thomas S. Kuhn [5] ein Satz allgemein anerkannter wissenschaftlicher Leistungen, der für eine gewisse Zeit einer Gemeinschaft von Fachleuten maßgebende Probleme und Lösungen liefert

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 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).

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

Schnittstelle *interface*, hier: 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.

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 großenteils (implizit) auf einer Umsetzung dieser fünf Prinzipien.

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

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 „kleinsten 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.

Trennung der Zuständigkeiten *separation of concerns*, grundlegendes Prinzip für ↑Modularisierung, u.a. durch ↑Kapselung und Aufteilung eines Gesamtsystems in ↑Schichten.

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.

YAGNI „*You ain't gonna need it*", (nicht nur) in der angelsächsischen Programmierwelt verbreitetes Akronym und wichtige Regel für die Programmierung. Zielt darauf ab, jeweils nur das zu implementieren, was im Augenblick wichtig ist oder von dem zweifellos klar ist, dass es in Kürze gebraucht wird. Versuch, durch einen pragmatischen Ansatz ein „zu viel" an Abstraktion zu vermeiden. Das zugrundeliegende Problem, das damit angegangen werden soll: Prognosen (hier: bzgl. der zukünftigen Anforderungen an eine bestimmte Software) sind schwierig, insbesondere wenn sie die Zukunft betreffen (vgl. ↑Kristallkugel).

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] Melvin E. Conway. How do committees invent? *Datamation* 14.4 (1968), S. 28–31.

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

[5] Thomas S. Kuhn. *Die Struktur wissenschaftlicher Revolutionen*. Frankfurt am Main: Suhrkamp, 1976.