Verteilte Systeme und Komponenten

Zusammenfassung Frühlingssemester 2018

Patrick Bucher

07.06.2018

Inhaltsverzeichnis

1 Komponenten

Herkunft: *componere* (lat.) = zusammensetzen

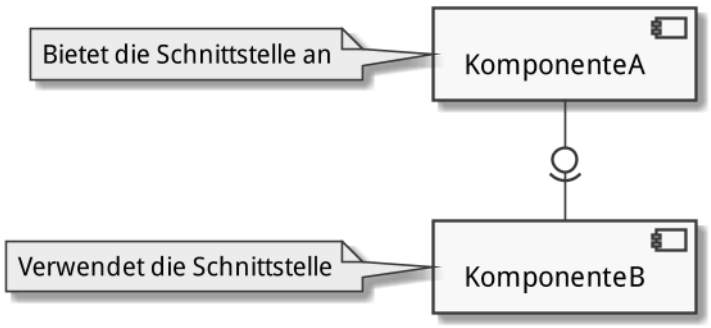

Abbildung 1: Komponentendiagramm (UML2)

1.1 Begriffe und Architekturen

1.1.1 Der Komponentenbegrif

- Definition: Eine Software-Komponente
  1. ist ein Software-Element
  2. passt zu einem bestimmten Komponentenmodell
  3. folgt einem bestimmten *Composition Standard*
  4. kann ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden
- Eigenschaften: Software-Komponenten
  1. sind eigenständig ausführbare Softwareeinheiten
  2. sind über ihre Schnittstellen austauschbar definiert
  3. lassen sich unabhängig voneinander entwickeln
  4. können kunden- und anwendungsspezifisch oder anwendungsneutral und wiederverwendbar sein

    - COTS (Commercial off-the-shelf): Software «von der Stange»
  5. können installiert und deployed werden
  6. können hierarchisch verschachtelt sein
- Komponentenmodelle
  - sind konkrete Ausprägungen des Paradigmas der komponentenbasierten Entwicklung
  - definieren die genaue Form und Eigenschaften einer Komponente
  - definieren einen *Interaction Standard*
    * wie können die Komponenten miteinander über Schnittstellen kommunizieren (Schnittstellenstandard)
    * wie werden die Abhängigkeiten der Komponenten voneinander festgelegt
      · von der Komponente verlange Abhängigkeiten: *Required Interfaces*
      · von der Komponente angebotene Abhängigkeiten: *Provided Interfaces*
  - definieren einen *Composition Standard*
    * wie werden die Komponenten zu grösseren Einheiten zusammengefügt
    * wie werden die Komponenten ausgeliefert (Deployment)
- Beispiele verbreiteter Komponentenmodelle:
  - Microsoft .NET
  - EJB (Enterprise Java Beans)
  - OSGi (Open Services Gateway Initiative)
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed Component Object Model)

1.1.2 Der Nutzen von Komponenten

- Packaging: *Reuse Benefits*
  - Komplexität durch Aufteilung reduzieren (*Divide and Conquer*)
  - Wiederverwendung statt Eigenentwicklung spart Entwicklungszeit und Testaufwand
  - erhöhte Konsistenz durch Verwendung von Standardkomponenten
  - Möglichkeit zur Verwendung bestmöglichster Komponente auf dem Markt
- Service: *Interface Benefits*
  - erhöhte Produktivität durch Zusammenfügen bestehender Komponenten
  - erhöhte Qualität aufgrund präziser Spezifikationen und vorgetesteter Software
- Integrity: *Replacement Benefits*
  - erweiterbare Spezifikation durch inkrementelle Entwicklung und inkrementelles Testing
  - parallele und verteilte Entwicklung durch präzise Spezifizierung und Abhängigkeitsverwaltung
  - Kapselung begrenzt Auswirkungen von Änderungen und verbessert so wie Wartbarkeit

1.1.3 Der Entwurf mit Komponenten

- Komponentenbasierte Enwicklung

  - steigende Komplexität von Systemen, Protokollen und Anwendungsszenarien
  - Eigenentwicklung wegen Wirtschaftlichkeit und Sicherheit nicht ratsam
  - Konstruktion von Software aus bestehenden Komponenten immer wichtiger
  - Anforderungen (aufgrund mehrmaliger Anwendung) an Komponenten höher als an reguläre Software
- Praktische Eigenschaften
  - Einsatz einer Komponente erfordert nur Kenntnisse deren Schnittstelle
  - Komponenten mit gleicher Schnittstelle lassen sich gegeneinander austauschen
  - Komponententests sind Blackbox-Tests
  - Komponenten lassen sich unabhängig voneinander entwickeln
  - Komponenten fördern die Wiederverwendbarkeit
- Komponentenspezifikation
  - Export: angebotene/unterstützte Interfaces, die von anderen Komponenten genutzt werden können
  - Import: benötigte/verwendete Interfaces von anderen Komponenten
  - Kontext: Rahmenbedingungen für den Betrieb der Komponente
  - Verhalten der Komponente

1.1.4 Komponenten in Java

- Komponenten in Java SE
  - Komponenten als normale Klassen implementiert
  - Komponenten können, müssen sich aber nicht and die *Java Beans Specification* halten
    * Default-Konstruktor
    * Setter/Getter
    * Serialisierbarkeit
    * PropertyChange
    * Vetoable
    * Introspection
  - Weitergehende Komponentenmodelle in Java EE
    * Servlets
    * Enterprise Java Beans
- Austauschbarkeit
  - Die Austauschbarkeit von Komponenten wird durch den Einsatz von Schnittstellen erleichtert.
  - Schnittstellen werden als Java-`Interface` definiert und dokumentiert (JavaDoc).
  - Eine Komponente implementieren eine Schnittstelle als Klasse.
    * mehrere, alternative Implementierungen möglich
    * Austauschbarkeit über Schnittstellenreferenz möglich
  - Beispiel: API von JDBC (Java Database Connectivity)
    * von Sun/Oracle als API definiert
    * von vielen Herstellern implementiert (JDBC-Treiber für spezifische Datenbanksysteme)

        * Datenbankaustausch auf Basis von JDBC möglich
- Deployment
  - über `.jar`-Dateien (Java Archive): gezippte Verzeichnisstrukturen bestehend aus
    * kompilierten Klassen und Interfaces als `.class`-Dateien
    * Metadaten in `META-INF/manifest.mf`
    * optional weitere Ressourcen (z.B. Grafiken, Textdateien)
  - Deployment von Schnittstelle und Implementierung zum einfacheren Austausch häufig in getrennten `.jar`-Dateien mit Versionierung, Beispiel (fiktiv):
    * `jdbc-api-4.2.1.jar` enthält die Schnittstelle
    * `jdbc-mysql-3.2.1.jar` enthält die MySQL-Implementierung
    * `jdbc-postgres-4.5.7.jar` enthält die PostgreSQL-Implementierung
    * Versionierung idealserweise im Manifest und im Dateinamen (Konsistenz beachten!)

1.2 Schnittstellen

1.2.1 Begriff und Konzept

- Der Begriff *Schnittstelle* als Metapher
  - Beim Zerschneiden eines Apfels entstehen zwei spiegelsymmetrische Oberflächen.
  - Die Komponenten müssen so definiert werden, damit sie an der Schnittstelle zusammenpassen, als ob sie vorher auseinandergeschnitten worden wären.
  - Tatsächlich werden *Verbindungsstellen* erstellt, welche Kombinierbarkeit sicherstellen.
  - Eine Schnittstelle tut nichts und kann nichts.
  - Schnittstellen trennen nichts, sie verbinden etwas:
    * Komponenten untereinander (Programmschnittstellen)
    * Komponenten mit dem Benutzer
- Die Bedeutung von Schnittstellen (bei korrektem Gebrauch):
  1. machen Software leichter verständlich (man braucht nur die Schnittstelle und nicht die Implementierung zu kennen)
  2. helfen uns Abhängigkeiten zu reduzieren (Abhängigkeit nur von einer Schnittstelle, nicht von einer Implementierung)
  3. erleichtern die Wiederverwendbarkeit (bei der Verwendung bewährter Schnittstellen statt Eigenentwicklung)
- Die Beziehung zwischen Schnittstellen und Architektur:
  - System > Summe seiner Teile (Beziehungen zwischen den Teilen: durch Schnittstellen ermöglicht)
    * Schnittstellen & Beziehungen zwischen den Komponenten: wichtigste Architekturaspekte!
    * Mehrwert des Systems gegenüber Einzelkomponenten liegt in den Schnittstellen & Beziehungen der Komponenten zueinander
  - Spezialisten für Teilsysteme konzentrieren sich auf ihr Zeilproblem
    * Architekten halten das Gesamtsystem über Schnittstellen zusammen

        * Schnittstellen verbinden ein System mit der Aussenwelt und ermöglichen die Interaktion damit
- Kriterien für gute Schnittstellen
    1. Schnittstellen sollen *minimal* sein:
        - wenige Methoden (mit möglichst geringen Überschneidungen in ihren Aufgaben)
        - geringe Anzahl von Parameters
        - setzen möglichst keine oder nur wenige globale Daten voraus
    2. Schnittstellen sollen *einfach zu verstehen* sein
    3. Schnittstellen sollen *gut dokumentiert* sein

1.2.2 Dienstleistungsperspektive

- Die Schnittstelle als Vertrag:
    - Ein *Service Consumer* schliesst einen Vertrag mit einem *Service Provider* für eine *Dienstleistung* ab
- Design by Contract (DbC): Das Zusammenspiel zwischen den Komponenten wir mit einem Vertrag geregelt
    - *Preconditions*: Zusicherungen, die der Aufrufer einhalten muss
        * Nutzer: Prüfen der Vorbedingungen vor der Ausführung
        * Anbieter: Überprüfung mittels Assertions
    - *Postconditions*: Nachbedingungen, die der Aufgerufene garantiert
        * Nutzer: Überprüfung mittels Assertions
        * Anbieter: Prüfen der Nachbedingungen nach der Ausführung
    - *Invarianten*: Über alle Instanzen einer Klasse geltende Grundannahmen ab deren Erzeugung
        * Anbieter: Überprüfung mittels Assertions

1.2.3 Spezifikation von Schnittstellen

- Dokumentation von Schnittstellen
    - Umfang:
        * was ist wichtig für die Benutzung der Komponente
        * was muss der Programmierer versethen und beachten
    - Eigenschaften der Methoden:
        * Syntax (Rückgabewerte, Argumente, Typen, call by value/reference)
        * Semantik (was bewirkt die Methode)
        * Protokoll (synchron/asynchron)
        * Nichtfunktionale Eigenschaften (Performance, Robustheit, Verfügbarkeit)
    - Schnittstellen an der Systemgrenze fliessen in die Systemspezifikation ein
- öffentliche Schnittstellen werden als *API* bezeichnet (Application Programming Interface)
    - objektorientierte API (sprachabhängig, z.B. API der JSE)

  - REST-API (Representational State Transfer, sprach- und plattformunabhängig, datenzentriert)
  - Messaging-API (sprach- und plattformunabhängig, z.B. Push-Notifications für Mobile Apps)
  - dateibasierte API (Informationsaustausch, Konfigurationsdateien)

1.3 Modularisierung

Modul: in sich abgeschlossener Teil des Programmcodes, bestehend aus Verarbeitungsschritten und Datenstrukturen

1.3.1 Modulkonzept

- Kopplung und Kohäsion
  - Kopplung: Ausmass der Kommunikation zwischen Modulen
    - hohe Kopplung: grosse Abhängigkeit
    - Kopplung minimieren!
  - Kohäsion: Ausmass der Kommunikation innerhalb eines Moduls
    - gerine Kohäsion: geringer Zusammenhalt
    - Kohäsion maximieren!
  - Viele Module: Hohe Kopplung, geringe Kohäsion
  - Wenige Module: Geringe Kopplung, hohe Kohäsion
  - Idealer Kompromiss: Reduziert Gesamtkomplexität
- Arten von Modulen
  - Bibliothek: Sammlung oft verwendeter, thematisch zusammengehörender Funktionen (Datumsmodul, Mathematik-Modul, I/O-Modul)
  - Abstrakte Datentypen: Implementierung eines neuen Datentyps mit definierten Operationen (verkettete Liste, binärer Baum Hash-Tabelle)
  - Physische Systeme: Abgegrenztes Hardware-Modul (Ultraschallsensor, Anzeigemodul, Kommunikationsmodul)
  - Logisch-konzeptionelles System: Modellierung von Funktionalität auf hoher Abstraktionsstufe (Datenbankmodul, Bildverarbeitungsmodul, GUI-Framework)
- Entwurfskriterien
  - Zerlegbarkeit (*modular decomposability*): Teilprobleme können unabhängig voneinander gelöst werden
    - *Divide and Conquer*: Softwareproblem in weniger komplexe Teilprobleme zerlegen, sodass sie unabhängig voneinander bearbeitet werden können
    - Rekursive Zerlegung: Weitere Zerlegung von Teilproblemen
  - Kombinierbarkeit (*modular composability*): Module sind unabhängig voneinander wiederverwendbar
    - Module sollten möglichst frei kombinierbar sein und sich auch in anderen Umfeldern wieder einsetzen lassen

        * Zerlegbarkeit und Kombinierbarkeit sind unabhängig voneinander
    - Verständlichkeit: Module sind unabhängig voneinander verständlich
        * Der Code eines Moduls soll ohne Kenntnis anderer Module verstehbar sein
        * Module müssen unabhängig voneinander versteh- und wartbar sein
    - Stetigkeit: Änderungen der Spezifikation proportional zu Codeänderungen
        * Anforderungen können sich ändern, sollten sich aber nur auf ein Teilsystem auswirken
- Entwurfsprinzipien
    - lose Kopplung: schlanke Schnittstellen, Austausch nur des Nötigsten
    - starke Kohäsion: hoher Zusammenhalt innerhalb des Moduls
    - Geheimnisprinzip (*information hiding*): Modul nach aussen nur über dessen Schnittstellen bekannt
    - wenige Schnittstellen: zentrale Struktur mit minimaler Anzahl Schnittstellen
    - explizite Schnittstellen: Aufrufe und gemeinsam genutzte Daten sind im Code ersichtlich
- Vorgehen bei Modularisierung
    - Basiskonzepte: Kopplung & Kohäsion
    - Kriterien: Verständlichkeit, Kombinierbarkeit, Zerlegbarkeit, Stetigkeit
    - Modularten: Bibliotheken, abstrakte Datentypen, physische und logische Systeme
    - Prinzipien: geringe Kopplung, hohe Kohäsion, Geheimnisprinzip, wenige & explizite Schnittstellen
    - sinnvolle Modularisierung: eine der anspruchsvollsten Aufgaben der Informatik
- Parnas: *On the Criteria to be Used in Decomposing Systems into Modules* (1972)
    - Ziele der Modularisierung:
        1. Die Flexibilität und Verständlichkeit eines Systems verbessern
        2. Die Entwicklungszeit eines Systems reduzieren
    - Voraussetzung für modulares Programmieren:
        1. Ein Modul kann mit wenig Kenntnis des Codes eines anderen Moduls geschrieben werden.
        2. Module können neu zusammengesetzt und ersetzt werden, ohne dass das ganze System neu zusammengesetzt werden muss.
    - Nutzen der Modularisierung:
        * Verkürzung der Entwicklungszeit, da mehrere Teams gleichzeitig an je einem Modul arbeiten können und nur wenig Kommunikation zwischen ihnen nötig ist.
        * Erhöhte Flexibilität, da grössere Änderungen an einem Modul keine Änderungen in anderen Modulen zur Folge haben.
        * Bessere Verständlichkeit, da ein System nicht als ganzes, sondern Modul für Modul analysiert werden kann.
    - Ansätze der Modularisierung:
        1. *Flowchart-Analyse*: Jeder grosse Verarbeitungsschritt wird als Modul implementiert (konventionell).
        2. *Information Hiding*: Jede Design-Entscheidung wird in einem Modul versteckt (neuer Ansatz).
    - Interpretation:
        * Mit dem traditionellen Ansatz (*Flowchart-Analyse*) wird ein *Algorithmus* in einzelne

Verarbeitungsschritte zerlegt.
    * Mit dem neuen Ansatz (*Information Hiding*) werden die *Datenstrukturen* herausgearbeitet. (Datenstruktur = Design-Entscheidung)
    * Die einzelnen Schritte eines Algorithmus sind *nicht* beliebig austauschbar.
    * Datenstrukturen können abstrahiert und über ein einfaches Interface angeboten werden.

1.3.2 Layers, Tiers & Packages

- Layer
  - öffentliche Methoden eines tieferstehenden Layers B dürfen vom höherstehenden Layer A genutzt werden
  - Beispiel (Layers von oben nach unten): `A B C`
    * richtig: `A -> B`, `B -> C`
    * zulässig: `A -> C` (gefährlich: Umgehung einer API)
    * falsch: `C -> B`, `B -> A`, `C -> A` (von unten nach oben)
    * falsch: `A -> B -> C -> A` (zyklische Abhängigkeit)
  - *call*-Beziehung: ein höherstehender Layer verwendet Funktionalität eines tieferstehenden Layers
  - *use*-Bezehung: korrektes Verhalten von Layer A hängt von der korrekten Implementierung des Layers B ab (initialisiertes Device, aufgenommene Netzwerkverbindung, erstellte Datei)
- Tier: oft mit Layern verwechselt
  - Presentation Tier
  - Business Logic (Tier)
  - Data Tier
- Packages: Implementierung des Layer-Konzepts
  - abstrakt: UML
  - konkret: Java-Package

2 Entwicklungsprozess

2.1 Projektplanung

2.2 Source-Code-Management, Build und Dependency-Management

2.3 Build-Server

2.4 Integrations- und Systemtesting

2.5 Entwurfsmuster

2.6 Testing

2.7 Continuous Integration

2.8 Review

2.9 Konfigurationsmanagement

2.10 Deployment

- Beim Deployment geht es um folgende Aspekte:
  1. Verteilung von Software und Dokumentation per Datenträger oder Web
  2. Installation: Software auf Zielsystem kopieren
  3. Konfiguration: Einstellung der Software auf jeweilige Anforderungen
  4. Organisation: Informatin, Schulung, Support, Planung, Produktion
- Wann findet Deployment statt?
  - Am Ende eines (erfolgreichen) Projekts.
  - Bei iterativen Entwicklungsmodellen öfters (Continuous Delivery erfordert Continuous Deployment)
  - Fortlaufendes Deployment einzelner Iterationsergebnisse (Build, Sprint, Meilenstein) auf Entwicklungs- und Testsysteme (Staging)
- Die Deployment-Dokumentation umfasst:
  - Technische Aspekte
    - Deploymentdiagramme: Zuordnung der Komponenten zu Systemen/Hardware (Zeigt Nodes, Komponenten pro Node und deren Verbindungen miteinander)
    - Installations- und Deinstallationsprogramme bzw. -skripte
    - Konfigurationen (Standardkonfiguration, Beispiele, kunden- und umgebungsspezifisch)
    - Installationsmedium/Bezugsquelle (physischer Datenträger, URL, Login)
    - Repositories zur Ablage der Binaries
  - Organisatorische Aspekte

        * Konfigurationsmanagement: Aus welchen Komponenten wird der Release gebildet?
        * Installations- und Bedienungsanleitung
        * Erwartungsmanagement: Welche Funktionalität ist vorhanden?
        * Bereitstellung von Support (intern/extern, 1st Level/2nd Level etc.)
- Installation und Deinstallation
    - Möglichst automatisierbar und dadurch reproduzierbar
    - Vollständige und saubere Deinstallation muss möglich sein
    - Vollautomatische Softwareverteilung (setzt bestimmte Merkmale voraus)
    - Unterschiedliche Bedürfnisse nach Benutzergruppe:
        * Endbenutzer: Interaktive Installation mit Setup-Wizard
        * Administrator: Möglichst automatisierbare Installation
        * Entwickler/Tester: Auf eigene Bedürfnisse zugeschnittene Installation
- Konfiguration von Anwendungen
    - Zielkonflikt: Software soll out-of-the-box laufen, aber möglichst auf die eigenen Bedürfnisse anpassbar sein (am liebsten out-of-the-box auf eigene Bedürfnisse konfiguriert)
    - Konfigurationsmanagement: Wer hat welche Version, welche Lizenzen, welche Umgebung? Welche Kombinationen sind lauffähig? Welche Update-Schritte funktionieren? Welche Szenarien wurden getestet?
- Deployment-Manuals
    - Installationsanleitung (möglichst klein mit automatisierter Installation)
        * Gehen auf verschiedene Konfigurationsvarianten und Voraussetzungen (Hardware/Software) ein
        * Behandelt einzuhaltende Abfolgen
    - Bedienungsanleitung (Release Notes)
        * neue Funktionen
        * neue Bedingungen zum Betreiben der Software
        * neue Datenformate und Protokolle
        * Manuell aus verschiedenen Quellen zusammengestellt: Commit-Messages, Bugfix-Meldungen, Feature-Beschriebe (User Stories) etc.
- Deployment-Support: Muss in jedem Fall bereitgestellt werden!
- Release und Versionierung
    - Eindeutige Versionsbezeichnung (technische Version, Tag im Versionskontrollsystem, evtl. zusätzliche «Marketing-Versionierung»)
    - Semantic Versioning: x.y.z Semver.org
        * x: Major (bei inkompatiblen Änderungen hochgezählt)
        * y: Minor (beim Hinzufügen rückwärtskompatibler Funktionalität hochgezählt)
        * z: Patch (bei Fehlerkorrekturen hochgezählt)
    - Ein Blick auf die neue Versionsnummer soll Klarheit geben, was sich geändert hat.
    - Zeitbasierte Modelle: Ubuntu, Gnome: alle 6 Monate, Arch: monatlicher Snapshot
- Technisches Deployment: Beispiel Java
    - Verteilung einzelner `.class`-Dateien: inakzeptabel, fehleranfällig
    - Verteilung von `.jar`-Archiven (Java Archive): gezippte `.class`-Dateien mit zusätzlichen Ressourcen und Meta-Daten (`META-INF/MANIFEST.MF`, u.a. für `CLASSPATH`-Angaben)
        * `.war`-Dateien (Web Archive) für Webcontainer (`META-INF/web.xml`)

    * `.ear`-Dateien (Enterprise Archive) für Applikationsserver (`META-INF/application.xml`)
  – Einzelne (kombiniert zu Shadow-Archiven) oder mehrere `.jar`-Dateien (erfordert `CLASSPATH`-Angabe)
  – Zusätzliche `.jar`-Dateien für Quellcode und Dokumentation möglich:
    * `foobar.jar`: Ausführbar (`.class`-Dateien)
    * `foobar-doc.jar`: Dokumentation (JavaDoc)
    * `foobar-src.jar`: Quellcode (`.java`-Dateien), v.a. bei Open Source

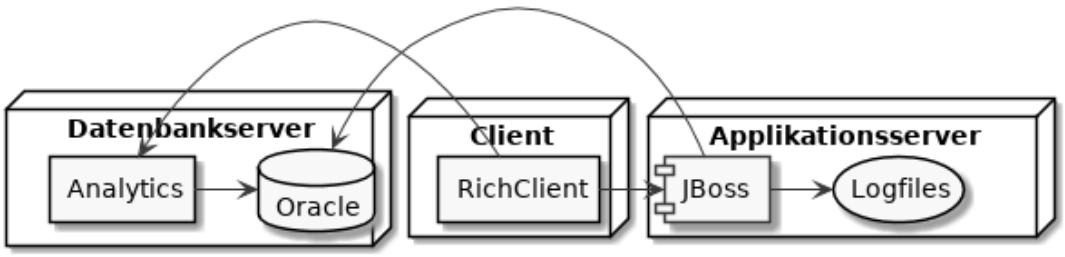

Abbildung 2: Deploymentdiagramm

2.11 Code-Qualität

2.11.1 Kommentare

- Schlechter Code soll nicht kommentiert sondern umgeschrieben werden.
  – Kommentare dienen oft als Ausrede für schlechten Code.
  – Selbsterklärender Code ist besser als jeder Kommentar.
  – Gute Namen ersparen viele Kommentare.
- Kommentare sind kein Qualitätsmerkmal sondern ein (manchmal) notwendiges Übel.
  – Code wird geändert, ohne dass die Kommentare nachgeführt werden.
  – Die Wahrheit liegt im Code, Kommentare können «lügen».
- Akzeptable Kommentare
  – Copyright (Lizenzbedingungen)
  – temporäre `TODO`/`FIXME`-Einträge
  – Kommentare zur Hervorhebung unauffälliger aber wichtiger Dinge
  – zusätzliche Erklärungen zur Absicht des Codes
  – Warnungen
- Schlechte Kommentare
  – Redundante Kommentare: `file.save(); // save the file`
  – Irreführende Kommentare: `extension = ".xml"; // txt file`
  – erzwungene Kommentare: JavaDoc für Getter- und Setter-Methoden
  – Tagebuch- und Changelog-Kommentare: Versionskontrolle verwenden
  – Positionsbezeichner und Banner: `// 3) now save the changes`
  – Zuschreibungen und Nebenbemerkungen: `created by XYZ (xyz@foo.com)`

  - Auskommentierter Code: Kann dank Versionskontrolle gelöscht werden
  - HTML-formatierte Kommentare: Im Code schlecht lesbar
  - Zu viele und unnötige Informationen: `// written on my balcony at 5:00 am`

2.11.2 Namensgebung

- Namen von Klassen sollten gut überlegt sein, da sie ihn lange behalten.
  - Noch wichtiger bei Interfaces!
- Kriterien für gute Namen:
  - zweckbeschreibend (schlecht: `ThingDoer`, gut: `TreeWalker`)
  - korrekt und ohne Fehlinformationen (schlecht: `ConnectionSingleton` wenn gar kein Singleton implementiert wird)
  - differenzierend
    - schlecht: `int a; float b; char c;`
    - gut: `int divisor; float dividend; char decimalSeparator;`
  - gut aussprechbar und suchbar sein
    - schlecht: `int n_pâtés_mangées; String достопримечательность; float lOlOlOlOlOlOfAc7Or; double schnäderegägs;`
    - gut: `int pates_eaten; String attraction; float lolFactor; double babble;`
  - möglichst keine Codierungen enthalten
    - schlecht: `int range_0x00_0xff; int clr00ff00;`
    - gut: `int rangeFirstByte; int rgbGreen;`
- Heuristiken zur Namensgebung:
  1. Beschreibende Namen wählen
  2. Namen passend zur Abstraktionsebene wählen
  3. Standardnomenklatur verwenden
  4. Eindeutige Namen wählen
  5. Namenlänge abhängig von Geltungsbereich
  6. Codierungen vermeiden
  7. Nebeneffekte in Namen miteinbeziehen

2.11.3 Funktionen

- Funktionen sollen klein sein.
  - Faustregel: Eine Bildschirmhöhe nicht überschreiten (mit vernünftiger Schriftgrösse)
  - Vorteil: Kleine Funktionen sind schneller verständlich.
  - Konsequenz: Es gibt mehr Funktionen und evtl. auch mehr Klassen, da Klassen nicht zu viele Methoden haben sollen. Dies wirkt sich positiv auf die Testbarkeit aus.
- Jede Funktion hat nur eine Aufgabe, welche sie gut erledigt (Unix-Philosophie).
  - Finden der Aufgabe: mit einen «to»-Satz: «to sort an array», «to establish a connection»
  - Abschnitte in Funktionen deuten auf die Verletzung dieses Prinzips hin.

- Jede Funktion arbeitet auf nur einer Abstraktionsebene (*Single Level of Abstraction*).
  - Eine Funktion, die Zeilen zählt, sollte sich nicht mit Encodings befassen.
  - Eine Funktion, die Zahlen rundet, sollte sich nicht mit Little- und Big-Endian kümmern.
  - Verletzung dieses Prinzips: Codeerosion/Zerfall (schlechter Code als Magnet für mehr schlechten Code)
- Die `switch`-Anweisung sollte vermieden werden.
  - `switch` deutet auf mehrere Aufgaben hin (verletzt Single-Responsibility-Prinzip)
  - `switch` muss bei jeder Erweiterung angepasst werden (verletzt Open-Closed-Prinzip)
  - `switch`-Konstrukte treten im Code oft mehrmals auf (verletzt Dont-Repeat-Yourself-Prinzip)
  - Lösung: Polymorphe Konstrukte (z.B. Strategy-Pattern, Funktionsreferenz)
- Die Anzahl der Funktionsargumente sollte klein gehalten werden.
  - Vertauschungsgefahr bereits ab zwei Argumenten
  - Lesbarkeit verschlechtert sich mit Anzahl Funktionsargumente (Zeilenumbrüche oder horizontales Scrolling)
  - Je mehr Funktionsargumente übergeben werden müssen, desto eher wird eines falsch gesetzt.
    - Ausprobieren von Kombinationen ist die Folge.
    - Die Dokumentation muss konsultiert werden.
  - Die eindimensionale Metrik aus *Clean Code* («je weniger Funktionsargumente, desto besser») ist mit Vorsicht zu geniessen!
    - `summe = addiere(summand1, summand2);` (strukturierte bzw. funktionale Lösung)
    - `summe = summand1.addiere(summand2);` (objektorientierte Lösung)
    - `summierer.a = summand1; summierer.b = summand2; summe = summierer.summiere()`
    - Bei der dritten «Lösung» gibt es *keine* Funktionsparameter, sie ist aber die denkbar schlechteste, da sie einen bestimmten Kontext voraussetzt.
  - Es gibt Funktionen, die *aus fachlicher Sicht* viele Parameter benötigen:
    - `Point p1 = new Point(13, 27);` (x- und y-Koordinate)
    - `Color salmon = new RGB(255, 153, 153);` (RGB-Farbe)
    - `Color translucentSalmon = new RGBA(255, 153, 153, 0.5);` (mit Alphakanal)
    - `Color salmon = new RGBBuilder().red(255).green(153).blue(153).build()` («schöner» aber schwerer lesbar, gerade wenn eine ungebräuchliche Reihenfolge wie «BRG» oder «GBR» gewählt wird)
  - Heuristik: Die Funktion soll unter Beibehaltung der Parametertypen und Weglassung der Parameternamen noch verständlich sein.
    - Gut: `createPoint3D(int, int, int);` – x, y und z sind intuitiv
    - Schlecht: `createFile(String, int, boolean);` – Dateiname intuitiv, andere Parameter nicht
    - Besser: `createFileIfNotExists(String, Permissions);`
      - `boolean`-Flag als zusätzliche Funktion mit sprechendem Namen
      - `int`-Dateiberechtigungen (`0x664` für `rw-rw-r--`) als Parameterobjekt mit ent-

sprechendem Typnamen

- Auf Flag-Argumente sollte verzichtet werden; besser mehrere Funktionen mit sprechenden Namen.
  - `openFile("foo.txt", true)` – was bedeutet `true`?
  - `openFileCreateIfNotExists("foo.txt");`
- Auf ungewollte Nebeneffekte sollte verzichtet werden.
  - Beispiel: `checkPassword(username, password)` erstellt eine Session, wenn die Credentials korrekt sind.
  - Die Funktion führt eine zusätzliche, verborgene Aufgabe aus (Verletzung *Single Responsibility Principle*)
  - Nebeneffekte können zu Race-Conditions führen, was bei der Fehlersuche problematisch ist.
  - Nebeneffekte sollen über den Funktionsnamen «kommuniziert» werden (z.B. `tryLogin` für obiges Beispiel).
- Output-Argumente sollen vermieden werden.
  - Der Rückgabewert sollte das Ergebnis einer Funktion beinhalten.
  - Verletzung: `writeInto(buffer, text)` – Der Parameter `buffer` enthält das Ergebnis der Operation, nicht der Rückgabewert.
  - Bei objektorientierten Sprachen sind Output-Argumente einfach vermeidbar.
- Funktionen sollen entweder «etwas tun» (einen Seiteneffekt haben) oder «antworten» (Informationen von einem Objekt liefern), nie beides.
  - Beispiel `map.set(key, val)` liefert `boolean` zurück
    - `true`, wenn das Attribut gesetzt wurde
    - `false`, wenn es das Attribut `key` *nicht* gibt (nicht, wenn es nicht geklappt hat!)
    - Der Code ist schwer zu verstehen und kann zu Missverständnissen führen.
  - Rückgabewerte von Funktionen verleiten zum Aufruf innerhalb einer Bedingung:
    - `if (!map.set("x", 42))` – ist schwer zu interpretieren
  - Lösung: Aufteilung in zwei Methoden!
    1. `map.hasAttribute(key):boolean` (Rückgabewert)
    2. `map.setAttribute(key, val):void` (Seiteneffekt)
- Exceptions sind Fehlercodes vorzuziehen.
  - Trennung Programmablauf und Fehlerbehandlung
  - Fehlercodes verleiten zum Aufrufen von Funktionen an Orten, wo Bedingungen verlangt werden.
  - Fehlercodes können zu tief verschachteltem Code führen.
  - Fehlerbehandlung ist eine Aufgabe, eine Funktion sollte nur eine Aufgabe erfüllen.
    - Die eine Funktion wirft eine Exception.
    - Die andere Funktion beginnt mit `try` und ruft erstere auf.
- Eine gute Namensgebung ist wichtig, aber schwer.
  - Lange Namen sind sprechender, jedoch mühsamer in der Handhabung.
  - Faustregel (Rob Pike): Lange Namen für grosse Gültigkeitsbereiche, kurze Namen für kleine Gültigkeitsbereiche:
    - `absolutePathToFileSystemTableFile` ist für den globalen Scope sinnvoll.
    - `i` für den Arrayindex und `n` für die Anzahl Elemente sind in einem `for`-Loop völlig

ausreichend, da konventionell.

- Code Smells für Funktionen (*Clean Code*): «When it stinks, change it.»
  1. Zu viele Argumente: Können zu Verwechslung und Unübersichtlichkeit führen.
  2. Output-Argumente: Entsprechen nicht der Erwartung des Aufrufers.
  3. Flag-Argumente: Deuten auf Funktionen mit mehreren Aufgaben hin.
  4. Tote Funkeionen: Was nicht aufgerufen wird, soll gleich gelöscht werden.

2.11.4 Unit-Tests

- Grundlagen des Unit-Testings
  - Definition *Unit Test*: «A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behaviour of that unit of work.»
  - Unit-Tests geben ein schnelles Feedback, ob etwas grundsätzlich funktioniert.
  - Unit-Tests sind die Basis für jedes Refactoring: Was vorher funktioniert, muss auch nachher noch funktionieren.
- Qualität von Testcode
  - Testcode ist kein Wegwerfcode und sollte die gleich hohe Qualität wie der getestete Code haben (Namensgebung, Struktur, Verständlichkeit).
    - Testcode lebt oft länger als Produktivcode, da Produktivcode refactored wird, aber anschliessend immer noch die gleichen Testfälle bestehen muss.
    - Testcode ist auch eine Form von Dokumentation des Produktivcodes.
  - Testcode muss gut lesbar sein. Es soll mit möglichst wenig Code möglichst viel ausgesagt werden. Eigenschaften für guten Testcode sind:
    1. Klarheit
    2. Einfachheit
    3. Ausdrucksdichte
- Test Driven Development
  - Test- und Produktivcode werden in kurzen, aufeinanderfolgenden Zyklen (Sekunden bis Minuten) geschrieben. Der Testcode eilt dem Produktivcode immer nur wenig voraus.
  - Gesetze des Test Drive Development:
    1. Produktiver Code darf erst implementiert werden, wenn es dafür einen Unit-Test gibt.
    2. Dieser Unit-Test darf nur gerade so viel Code enthalten, dass er fehlerfrei kompiliert, aber der Test scheitert.
    3. Man ergänzt jeweils nur gerade so viel produktiven Code, bis dass der Test erfolgreich durchläuft.
- Aufbau und Organisation von Tests
  - Domänenspezifische Testsprache: durch Schreiben eigener Assert-Methoden
    - der Name der Assert-Methode soll möglichst aussagekräftig sein
  - Pro Testfall sollte möglichst nur eine Assertion verwendet werden
  - Pro Testfall sollte nur eine Sache getestet werden. Es gelten die Prinzipien:
    - SOC: Separation of Concerns (Verschiedene Klassen werden durch verschiedene

Tests abgedeckt.)
        * SRP: Single Responsibility Principle (Jeder Testfall kümmert sich um genau eine Sache.)
        * SLA: Single Level of Abstraction (Ein Testfall operiert nicht auf verschiedenen Abstraktionsstufen.)
    - Viele kleine Testfälle mit wenig Assertions erlauben eine gezieltere – und schnellere – Ausführung von Tests.
    - BOC-Pattern:
        * Build: Erstellen der Testdaten
        * Operate: Manipulieren der Testdaten
        * Check: Verifizieren der Ergebnisse
    - AAA-Pattern:
        * Arrange: Ausgangssituation schaffen
        * Act: Aktion ausführen
        * Assert: Ergebnis überprüfen
    - FIRST-Prinzip:
        * Fast: Tests sollen schnell sein.
        * Independent: Tests sollen unabhängig voneinander ausführbar sein.
        * Repeatable: Tests sollen immer auf jeder Umgebung ausführbar sein.
        * Self-Validating: Das Testergebnis muss sofort ersichtlich sein (failure/success).
        * Timely: Tests sollten rechtzeitig und möglichst vor dem Produktivcode geschrieben werden.
- Test-Heuristiken von Clean Code:
    1. Unzureichende Tests vermeiden: Eine möglichst hohe Testabdeckung anstreben.
    2. Coverage-Werkzeug verwenden: Nicht abgedeckte Codeteile ermitteln.
    3. Triviale Tests umsetzen: Nichts ist zu trivial, um nicht getestet zu werden.
    4. Ignorieren von Tests: Nur temporär und mit `@Ignore` statt auskommentieren.
    5. Grenzbedingungen testen: Damit der ganze Wertebereich abgedeckt wird.
    6. Fehler-Nachbarschaft testen: Wo ein Fehler ist, ist auch oft ein zweiter.
    7. Muster des Scheiterns analysieren: Anhang Gemeinsamkeiten auf Ursachen schliessen.
    8. Coverage im Fehlerfall analysieren: Branch möglicherweise aufgrund von Tippfehler verpasst.
    9. Schnelle Tests: Damit sie oft und gerne ausgeführt werden.

2.11.5 Weitere Massnahmen

- Reviews
    - Effizienteste Methode zur Verbesserung der Code-Qualität
    - Zu Beginn alleine oder in kleinen Teams, später mit mehreren Teilnehmern
    - In vertrauensvoller Atmosphäre, nicht als QS-Massnahme oder in Verbindung mit KPIs
- Weitergabe von Erfahrungen im informellen Austausch
    - Mit Snacks (5 Minuten) als Anreiz, vor/nach Vormittags-/Nachmittagspause
- Bemühungen für sauberen Code als Motivation und zur Erinnerung optisch präsent halten

- Werkzeuge: Checkstyle, PMD, Findbugs, SonarQube etc.
- Pfadfinderregel: Den Platz (Code) sauberer verlassen als man ihn angetroffen hat.

3 Verteilte Systeme

3.1 Socket-Kommunikation

3.2 Serialisierung

3.3 Message-Passing

3.4 Verteilung & Kommunikation: RMI

3.5 Uhrensynchronisation

3.6 Verteilung: Data Grid